DOI Target Solution Reference Architecture

Service Oriented Integration Center of Excellence
Chief Technology Officer Council (CTOC)

U.S. Department of the Interior
Office of the Chief Information Officer
1849 C Street NW
Washington, DC 22240

Version 1.2
January 24, 2007

Revision History

Version	Date	Author	Reviewer	Comments
0.1	10/22/04	CTOC		Original name – "DOI Solution Architecture"
0.2	5/22/06	CTOC		Completely new sections and reorganized
1.0	5/26/06	CTOC		Incorporated all comments from CTOC reviewers
1.1	7/21/06	CTOC		Incorporated comments from IEA reviewers and added an Executive Summary
1.2	1/24/07	CTOC		Changed name to DOI Target Solution Reference Architecture

Executive Summary

The DOI Target Solution Reference Architecture (TSRA) is the target logical solution architecture and service-oriented application reference architecture for the Department of the Interior (DOI). TSRA is intended to help the department and its bureaus attain their business needs using cost effective, usable, and maintainable enterprise applications. DOI plans to implement this unified solution architecture throughout the enterprise to achieve a flexible business model that meets the customers' demands with greatly reduced cost and much improved efficiency.

This document provides an architecture overview and policy for DOI use of solution architecture (SA) for information technology (IT) applications. It provides the service-oriented architecture framework and the IT building blocks upon which the future DOI complete solution will be developed. The TSRA summarizes the main conceptual elements and the relationships between these elements to aid a clear understanding of the DOI target architecture. It strives to achieve the following:

- Outline the enterprise solution architecture that will help the department and its bureaus achieve their business goals and IT Strategic Plans
- Facilitate early validation of the solution architecture's impact on DOI
- Define the role of solution architecture in making the DOI enterprise architecture more service oriented, creating more reusable assets
- Create a communication medium for the stakeholders: management, sponsors, users, architects, developers, and implementers
- Establish criteria and standards for product selection
- Provide a template for the vendors to validate their proposed solutions
- Provide a continuing path of analysis for EA blueprints

This TSRA document provides the relationship of the solution architecture (SA) with business, data, application, technology, [redacted]. It also explains how the solution architecture is driven by DOI business needs. Solution architecture and its associated application architecture address multiple development and deployment platforms. It applies Enterprise Application Integration (EAI) techniques in the application architecture. Consistent use of the TSRA will allow DOI to smoothly transition to a service-oriented architecture (SOA).

Appendix A of this document presents a Service Oriented Architecture (SOA) roadmap for DOI and discusses SOA patterns, [redacted]. Platform-specific reference architectures are presented in the .NET Application Reference Architecture and J2EE Application Reference Architecture documents.

Table of Contents

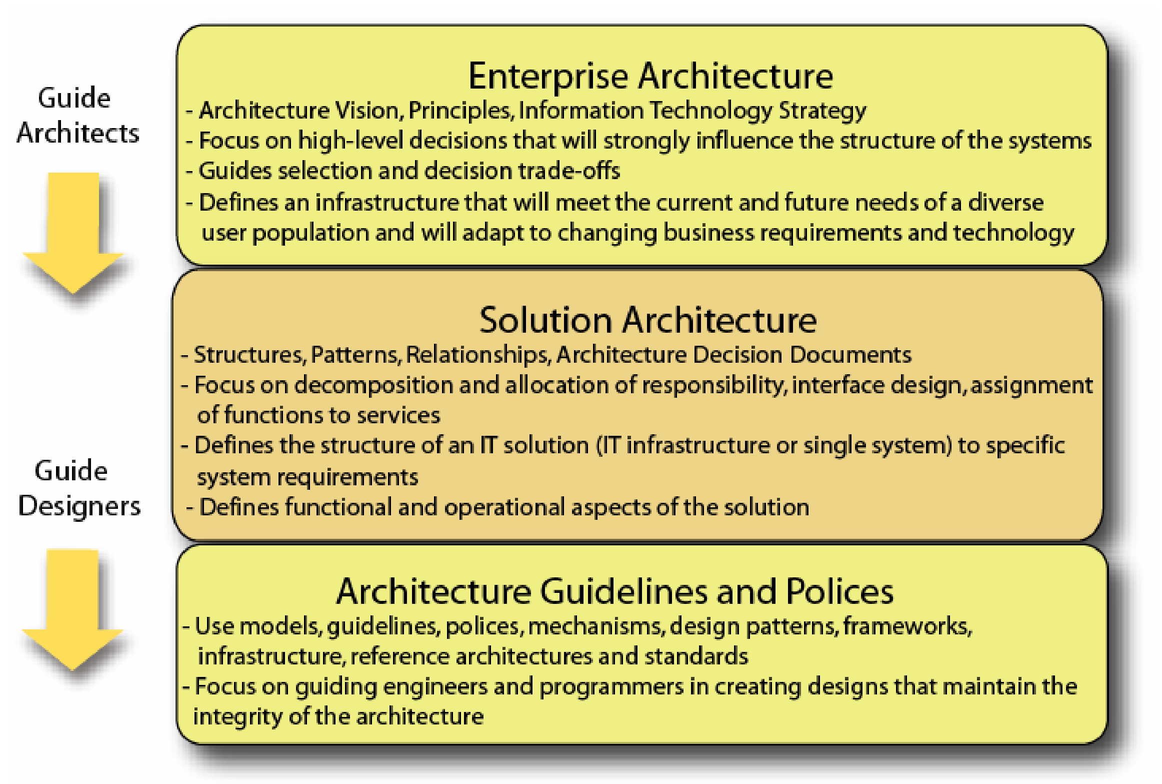
Table of Exhibits

1 Introduction

The DOI Target Solution Reference Architecture (TSRA) is the target logical solution architecture and service-oriented application reference architecture for the Department of the Interior (DOI). TSRA is intended to help the department and its bureaus attain their business needs using cost effective, usable, and maintainable enterprise applications. DOI plans to implement this unified solution architecture throughout the enterprise to achieve a flexible business model that meets the customers' demands with greatly reduced cost and much improved efficiency.

The DOI has developed and is implementing the Interior Enterprise Architecture (IEA) and is currently establishing the application architecture that complies with the IEA. As part of this effort, DOI is aligning with the TSRA the architecture domains that contribute to the IEA. This TSRA document provides the relationship of the solution architecture (SA) with business, data, application, technology, [redacted]ins. It also explains how the solution architecture is driven by DOI business needs. Solution architecture and its associated application architecture address multiple development and deployment platforms. It applies Enterprise Application Integration (EAI) techniques in the application architecture. Consistent use of the TSRA will allow DOI to smoothly transition to a service-oriented architecture (SOA).

Exhibit 1-1: Relationship of EA to SA


Guide Architects
Guide Designers
Enterprise Architecture
- Architecture Vision, Principles, Information Technology Strategy
- Focus on high-level decisions that will strongly influence the structure of the systems
- Guides selection and decision trade-offs
- Defines an infrastructure that will meet the current and future needs of a diverse user population and will adapt to changing business requirements and technology
Solution Architecture
- Structures, Patterns, Relationships, Architecture Decision Documents
- Focus on decomposition and allocation of responsibility, interface design, assignment of functions to services
- Defines the structure of an IT solution (IT infrastructure or single system) to specific system requirements
- Defines functional and operational aspects of the solution
Architecture Guidelines and Polices
- Use models, guidelines, polices, mechanisms, design patterns, frameworks, infrastructure, reference architectures and standards
- Focus on guiding engineers and programmers in creating designs that maintain the integrity of the architecture

This document presents a unified set of concepts that apply across all business applications within DOI. Adhering to these concepts will yield consistent and interoperable applications, enabling a service-oriented enterprise architecture. Ancillary documents provide scenarios, mappings, technologies, and other details. The ancillaries include *Interior Enterprise Architecture* and the *J2EE* and *.NET Application Reference Architecture* documents. Isolating the rapidly changing technologies into their respective documents allow them to be updated frequently and independently of this core architecture document. It is important to note that the principles and concepts presented here are not affected by technological changes.

This document is presented at a higher level of abstraction than the J2EE and .NET reference architecture documents. The reference architecture documents deal with how these technology standards impact the application architecture, concentrating on the physical technology concepts of specific J2EE and .NET products. This document concentrates on the logical building blocks of solutions and their applications. Its primary focus is on the structure of solutions and their resultant applications based on a logical partitioning of roles and responsibilities of the applications and infrastructure. Intelligently structuring applications allows a higher degree of integration with legacy systems, shorter cycle times, and flexibility to deal with rapidly changing technologies. This approach clearly positions the TSRA to support a service-oriented architecture.

1.1 Purpose

This document provides an architecture overview and policy for DOI use of solution architecture (SA) for information technology (IT) applications. It provides the service-oriented architecture framework and the IT building blocks upon which the future DOI complete solution will be developed. The TSRA summarizes the main conceptual elements and the relationships between these elements to aid a clear understanding of the DOI target architecture. It strives to achieve the following:

- Outline the enterprise solution architecture that will help the department and its bureaus achieve their business goals and IT Strategic Plans
- Facilitate early validation of the solution architecture's impact on DOI
- Define the role of solution architecture in making the DOI enterprise architecture more service oriented
- Create a communication medium for the stakeholders: management, sponsors, users, architects, developers, and implementers
- Establish criteria and standards for product selection
- Provide a template for the vendors to validate their proposed solutions
- Provide a continuing path of analysis for EA blueprints

The TSRA is intended to migrate DOI towards a service-oriented architecture (SOA), yielding the following benefits:

- More agility
- Tighter integration
- Higher quality
- Faster implementation

- Better skill leverage

In summary, solution architecture sets the context for creating DOI applications. Each sub-architecture provides a specific context to inform the overall application design. Business and application patterns provide common solutions to problems found across DOI applications. The reference architectures describe how to transform the application architecture for an implementation using a specific technology platform. They also give each application a significant head start by providing standard component designs so that the developers can concentrate on meeting the unique business requirements of the specific application, instead of re-inventing the wheel each time. Applications that comply with the TSRA will not only meet DOI business needs, they will be flexible, consistent, and efficient.

1.1.1 Scope

This document defines the DOI target solution architecture, which supports an application's business, data, application , technology , [redacted] While it describes the application architecture in detail, discussions of the other four architectures are beyond its scope.

1.2 Audience

This document is intended mainly for senior technical managers at DOI to foster a common understanding on IT governance. It may also benefit the requirements analysts, enterprise architects, solution architects, developers, and project managers who participate in solution and application development.

1.3 Document Organization

The document starts with an overview of enterprise and application architecture concepts and then describes the details of the sub-architectures. This chapter provides an introduction as well as the purpose, scope, audience, and organization of the document.

Chapter 2, Architecture Overview, presents the context of solution and application architecture within enterprise architecture. The following five chapters present each of the sub-architectures.

Chapter 3, Business Architecture, discusses the solution overview, use cases, and business needs, requirements, and patterns.

Chapter 4, Data Architecture, applies the concepts Subject Areas, Information Classes, and Conceptual and Logical Entities in the DOI Data Reference Model (DRM) to the TSRA. It also presents Entity Relationship Diagrams and Object Role Model diagrams.

Chapter 5, Application Architecture, begins with application patterns and non-functional requirements. It discusses architectural reference models, application interface types, and service-oriented architecture. It presents component and operational models, and then provides a methodology for making architectural decisions. It also discusses test-driven development and mapping the DOI Service Component Reference Model (SRM) to solution functionality.

Technology Architecture and the DOI Technology Reference Model (TRM) are reviewed in Chapter 6.

[redacted]

A brief synopsis of the steps that are taken when applying the TSRA is presented in Chapter 8, Using Solution Architecture.

Appendix A has a glossary and a list of abbreviations and acronyms used in the TSRA. Appendix B contains a list of all the artifacts discussed in the TSRA. A roadmap with pragmatic, concrete steps for transforming DOI applications to the target service-oriented architecture is presented in Appendix C.

1.4 References

[1] *Patterns for e-business: A Strategy for Reuse*, IBM Press, 2001

[2] *Service-Oriented Architecture: Concepts, Technology, and Design*, Thomas Erl, Prentice Hall, 2006

[3] *Enterprise Solution Patterns Using Microsoft .NET Version 2.0,* Microsoft Press, 2003

[4] *Application Architecture for .NET: Designing Applications and Services*, Microsoft Press, 2002

[5] *Improving .NET Application Performance and Scalability*, Microsoft Press, 2004

[6] *Smart Client Architecture and Design Guide*, Microsoft Press, 2005

[7] *Data Patterns*, Microsoft Press, 2005

[8] *Application Interoperability: Microsoft .NET and J2EE*, Microsoft Press, 2004

[9] *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design,* Terry Halpin, Morgan Kauffman, 2001

[10] “Inside Patterns”, Douglas Schmidt, Siemans AG, 1998, 1999

[11] “DOI Solution Architecture.doc” (v0.1), CTOC, 10/22/2004 [Version 0.1 of this document]

[12] “IBM e-business Pattern Integration White Paper”, U.S. Patent and Trademark Office, 11/25/2002

[13] “DOI .NET Reference Architecture” (v0.1), CTOC

[14] “DOI J2EE Reference Architecture” (v0.1), CTOC

[15] DOI Technology Reference Model v3.1 (TRM), CTOC, 5/1/2006

[16] Services and Component Based Architectures: A Strategic Guide for Implementing Distributed and Reusable Components and Services in the Federal Government, Version 3.5, Federal CIO Council, January 2005

[17] DOI Conceptual Architecture, IEA, May 2005

[18] Methodology for Business Transformation, IEA

[19] [redacted]

[20] [redacted]

2 Architecture Overview

This document uses the Interior Enterprise Architecture (IEA) documents and *Patterns for e-Business* [1] industry best practices to derive the DOI Solution Architecture (TSRA). The TSRA is also based on DOI's and the bureaus' strategy to standardize on commercial off-the-shelf (COTS) software components, to leverage existing J2EE and .NET technologies, and to integrate seamlessly with the existing infrastructure. One of the important goals of the TSRA is for DOI to migrate to a service-oriented architecture (SOA), gradually, with little or no disruption to the existing business operations. Additionally, it aims to promote cooperation and interoperation across bureaus and with other Federal Government agencies using secure, multi-channel solutions built on standards-based reusable services.

DOI and its Bureaus are leveraging e-Government strategies to make themselves more agile. This Solution Architecture (SA) initiative is a key step in improving DOI's customer service. The key business drivers behind this initiative are as follows:

- Provide consistent human and application interfaces throughout DOI
- Provide services that help reduce training costs
- Provide interfaces that are easy to use

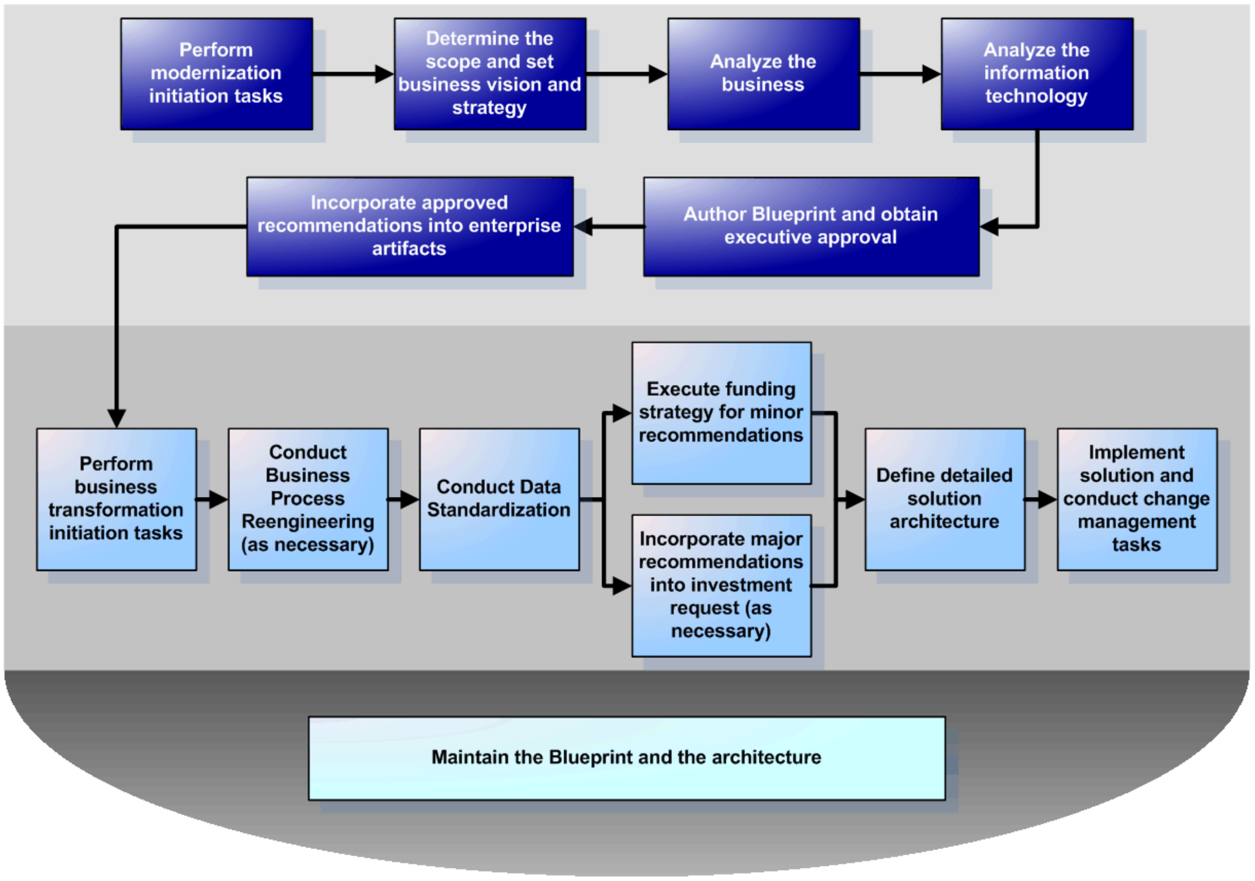
In addition, the department-wide initiative supports the goal of the Office of the Chief Information Officer (OCIO) to improve DOI's internal and external business practices using electronic services. It promotes solution consistency and simplifies vendor selection.

2.1 Enterprise Architecture

Enterprise Architecture (EA) is defined as a process and framework that leads to the development, implementation, maintenance, and use of a "blueprint" that explains and guides how an organization's business practices, information technology (IT), and information management elements work together to accomplish the mission. The EA must take into account the organization's business needs, performance measures, information, workflows, and processes. It must not be a set of technical decisions made in isolation of these other elements. In order to create the DOI EA, the Interior Enterprise Architecture (IEA) team leverages the Methodology for Business Transformation (MBT).

The MBT, shown in the visual below, consists of steps for creating a Modernization Blueprint for a business area, and then steps for implementing the business transformation outlined in the Modernization Blueprint. Within the first five steps of the MBT, the analysts are engaged in analysis of the business area's stakeholders, strategy, processes, and legacy technologies. Additionally, the analysts are working with the solution architects to develop a high level solution architecture to include in the Modernization Blueprint. Ultimately, the Modernization Blueprint include an assessment of the current state and recommendations for an improved performance, process, products, and solutions environment for the business area in focus.

Exhibit 2-1: IEA Methodology for Business Transformation


Perform modernization initiation tasks
Determine the scope and set business vision and strategy
Analyze the business
Analyze the information technology
Author Blueprint and obtain executive approval
Incorporate approved recommendations into enterprise artifacts
Perform business transformation initiation tasks
Conduct Business Process Reengineering (as necessary)
Conduct Data Standardization
Execute funding strategy for minor recommendations
Incorporate major recommendations into investment request (as necessary)
Define detailed solution architecture
Implement solution and conduct change management tasks
Maintain the Blueprint and the architecture

More information about the MBT can be found at www.doi.gov/ocio/architecture/mbt. The MBT is currently being extended to version 1.5 which will include more overt connections between the Enterprise Architecture and Solution Architecture teams and their artifacts.

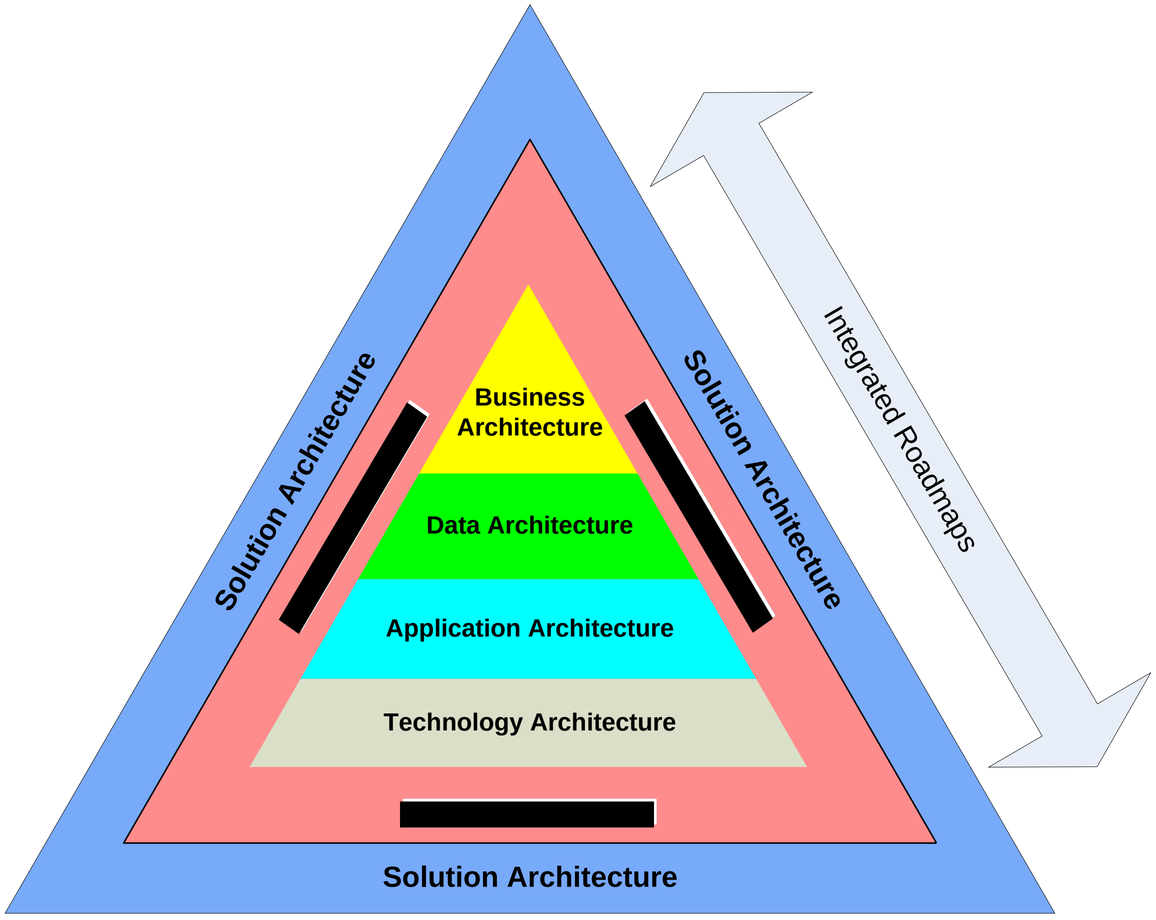
The scope of EA is the entire enterprise. The intent of EA (and its sub-architectures) is to describe and manage all of the enterprise business practices and applications that provide business solutions meeting the enterprise goals. With regard to EA, the DOI is creating the Interior Enterprise Architecture (IEA), envisioned to optimize service delivery by helping to effectively integrate and leverage existing investments across the DOI. The IEA is aimed at supporting investment decisions, improving the management of IT assets, and establishing a plan for transitioning toward target architectures within the organization.

Solution Architecture plays an important role in achieving these goals by enabling the transition to a Service-Oriented Architecture (SOA). The enterprise solution and application architecture are critical in the optimization of both hardware and software resources. One way that they do this is by providing a layer of abstraction between the logical solution and application architecture (as described in this document) and the physical technical architecture (described in the Reference Architectures [13], [14] and the Technical Reference Model (TRM) [15]). In other words, the solution and application architectures provide a *logical perspective* of an application within the context of its solution, describing *what* different elements of an application do in terms of their roles and responsibility. In contrast, the technology architecture provides the *physical perspective* of the application describing *how* those logical elements are constructed and connected using specific technologies within its solution. This separation provides some important benefits:

- Allows similar applications to be deployed on the same set of hardware resources because the structure of the solutions and applications is consistent.
- Allows the same application architecture to be applied across a variety of technology platforms, such as .NET and J2EE and still have a consistent structure.
- Allows optimization of the DOI data center strategy and its business infrastructures
- Allows for the EA analysts to engage the business community directly and develop a target business vision and agreement on the conceptual target systems environment
- Allows the SA analysts to work from business agreements generated in Steps 1-3 of the MBT and to develop a solutions architecture in Step 4 of the MBT in the Modernization Blueprint

The separation of the logical and physical aspects of the architecture is an example of the fundamental architectural principle of *separation of concerns*, in other words, separating issues to keep independent things independent (uncoupled from each other). The separation of concerns is applied to the enterprise to identify five separate constituent architectures that make up SA, illustrated in Exhibit 2-2. This separation of concerns means that each of the different aspects of the enterprise can be represented by an architecture that addresses its particular concerns. By being more focused, a particular architecture can be more effective at communicating to a particular audience.

Exhibit 2-2: BDATS Architecture



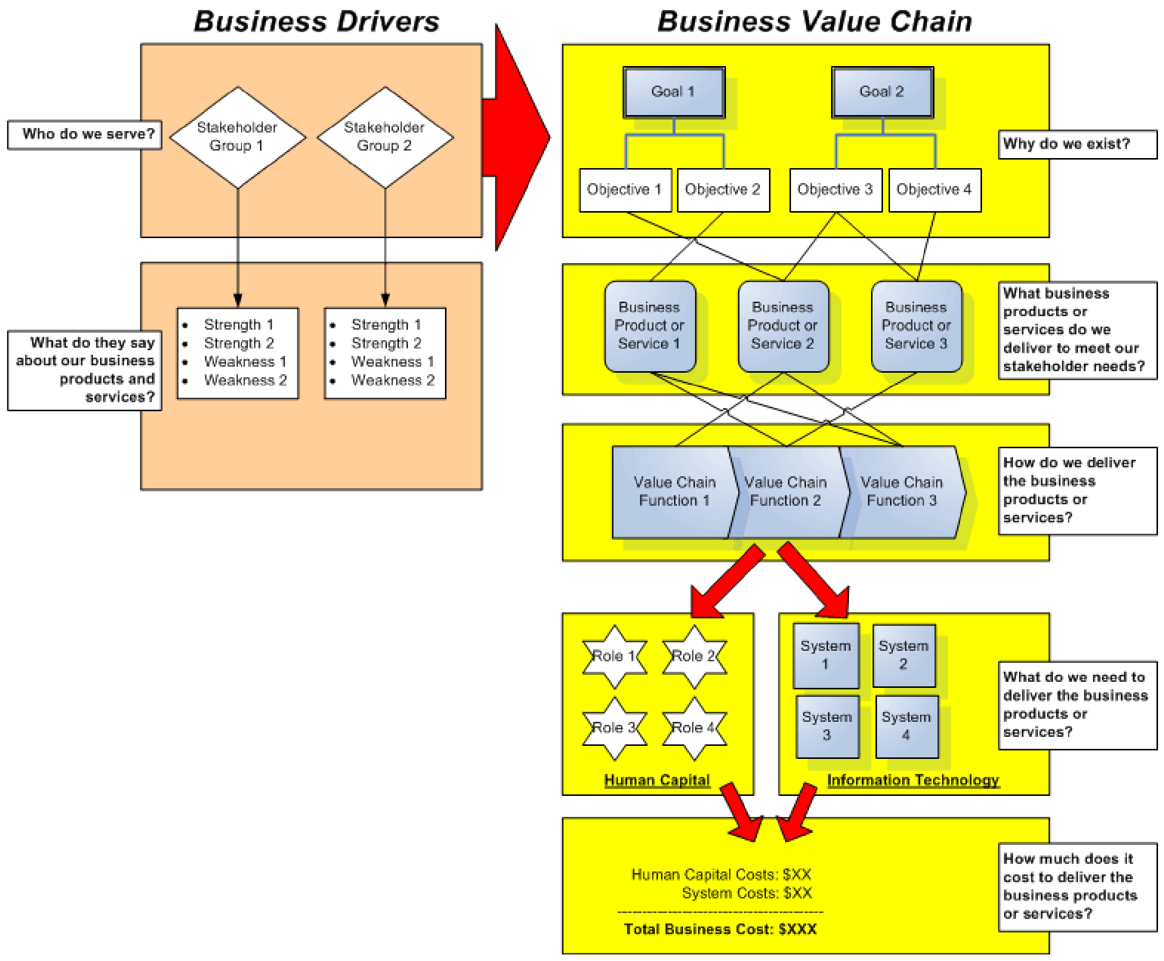
The TSRA identifies five primary constituent architecture domains. Collectively known as "BDATS", these sub-architectures represent the separate EA viewpoints of Business Architecture, Data Architecture, Applications Architecture, Technology Architecture and [redacted]

1. The Business Architecture domain defines the DOI's business and the information used in conducting business. It addresses functions and processes performed and services provided.
2. The Data Architecture domain defines the major kinds of data needed to support the DOI's business.
3. The Applications Architecture domain defines the structure of applications needed to manage the data and support the DOI's business functions.
4. The Technology Architecture domain defines the technology platforms needed to provide an appropriate quality of service and secure environment for the applications that manage the data and support the business functions.
5. [redacted]

6. **Solution Architecture** defines the end-to-end IT solution to a particular business problem. It covers both functional aspects as well as operational aspects. This is the basic foundation tying the rest of architectures together to provide the right solutions for solve DOI business challenges.

DOI's features a segmented enterprise architecture that includes the architecture domains mentioned above. Each segment in the DOI EA equates to a business area within the DOI enterprise. Within each business area or segment, the MBT is used to develop the information that is necessary to transform that business area. All of this information can be mapped back to the architecture domains listed above. Specifically, the following exhibit shows the line of sight that illustrates how information is developed and synthesized on the way to developing a target solutions architecture.

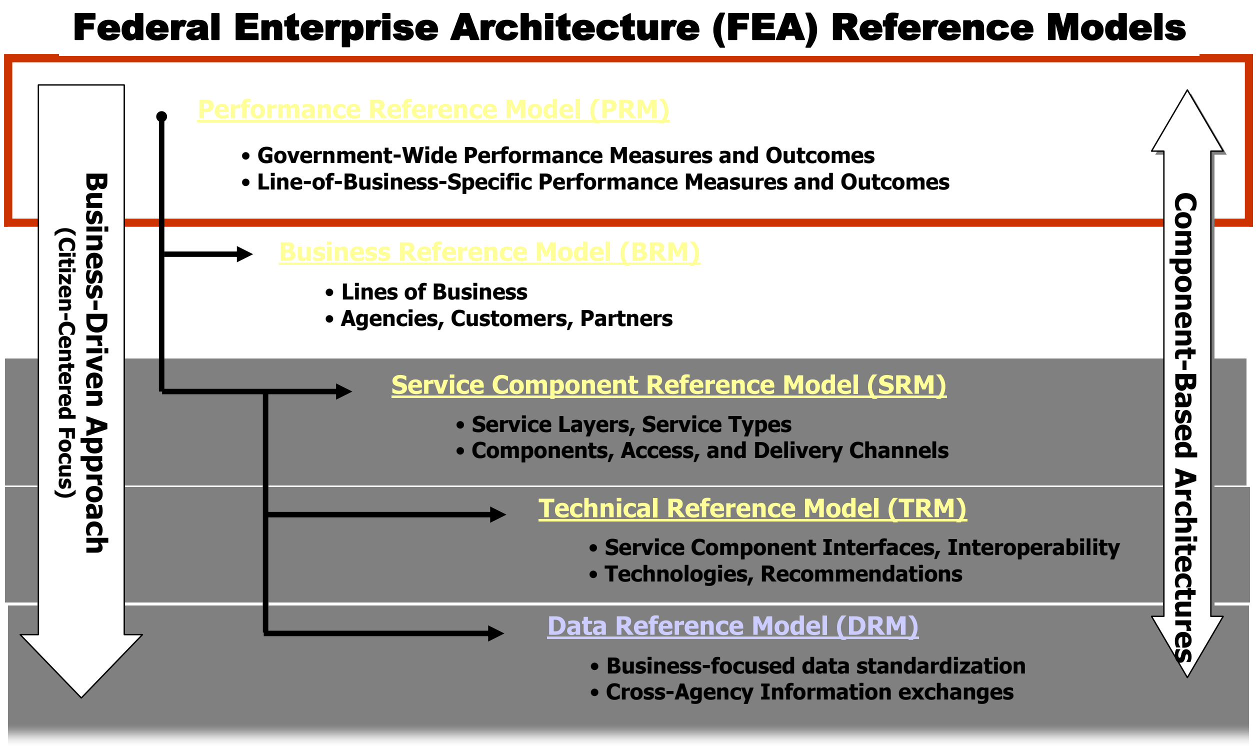
Exhibit 2-3: Applying the MBT



Each solution-level project needs to be viewed in terms of its relationship to the IEA. The IEA can be used to position individual solution-level projects within the larger DOI enterprise context. This practice conforms to the architectural principle that consideration of solution aspects within the next larger context, so that relationships among architectural elements can be fully understood and appreciated, is essential to the efficient and effective management of Information Technology. Connections and concepts in the IEA can be mapped to activities carried out and artifacts created and used in the DOI/Bureaus Information System Development Lifecycle Management (SDLM) processes.

The IEA relates FEA reference models (Exhibit 2-4) to DOI Enterprise Architecture. Each FEA reference model provides an OMB prescribed taxonomy for the overall Enterprise Architecture. For example, the BRM is a taxonomy for business lines and functions within the enterprise , while the SRM is a taxonomy for services within the enterprise.

Exhibit 2-4: Federal Enterprise Architecture (FEA) Reference Models


Federal Enterprise Architecture (FEA) Reference Models
Business-Driven Approach
(Citizen-Centered Focus)
Performance Reference Model (PRM)
• Government-Wide Performance Measures and Outcomes
• Line-of-Business-Specific Performance Measures and Outcomes
Business Reference Model (BRM)
• Lines of Business
• Agencies, Customers, Partners
Service Component Reference Model (SRM)
• Service Layers, Service Types
• Components, Access, and Delivery Channels
Technical Reference Model (TRM)
• Service Component Interfaces, Interoperability
• Technologies, Recommendations
Data Reference Model (DRM)
• Business-focused data standardization
• Cross-Agency Information exchanges
Component-Based Architectures

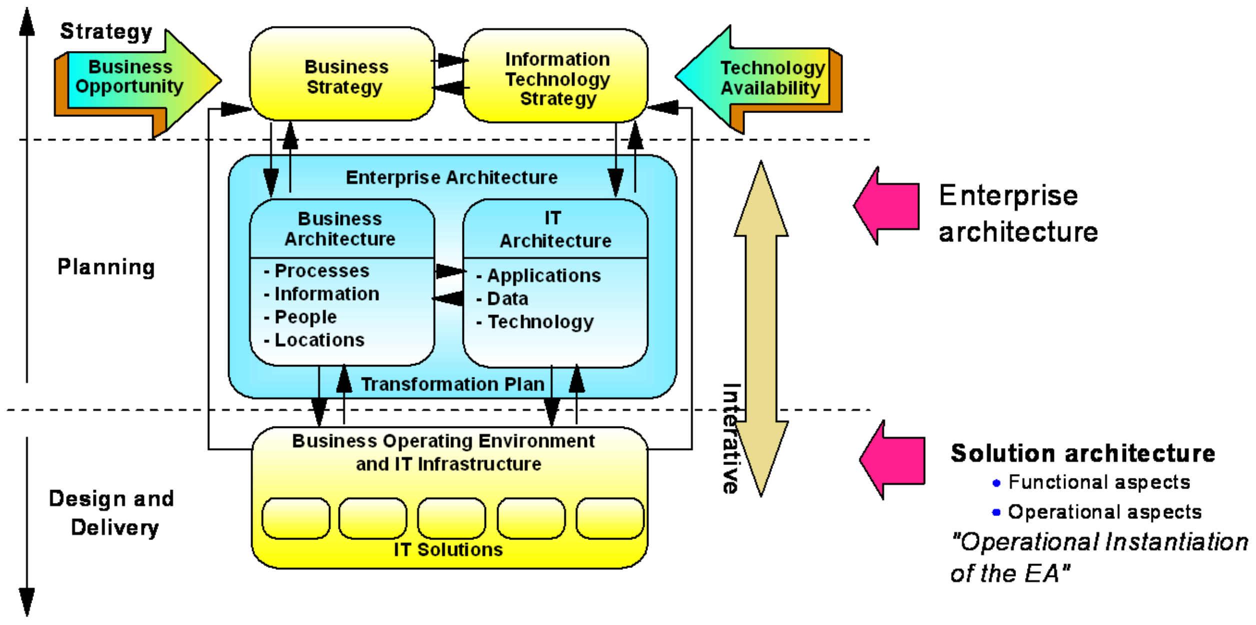
The IEA maps the FEA Business Reference Model (BRM) to DOI systems and DOI activity based costing codes. The Department Enterprise Architecture Repository (DEAR) and the Bureau Enterprise Architecture Repository (BEAR) provide the detail mapping description for each of the FEA reference models and how the relate to DOI investments, systems, and business activities. Throughout the MBT and throughout the development of the solution architecture, there are DEAR reports that are valuable inputs for the analysts as they do their work. These DEAR reports are available through the IEA website.

The Federal Enterprise Architecture (FEA) Business Reference Model identifies lines of business and some of these lines of business will be studied as part of the DOI Enterprise Architecture. For instance, the Law Enforcement line of business in the FEA BRM has been studied as part of the DOI EA and a Modernization Blueprint for this line of business has been created The following visual represents the lines of business and sub-functions from the FEA BRM that are active in the Department of the Interior and that will be affected by the [redacted]

[redacted]

Exhibit 2-6 identifies the relationship between Department and Bureau Enterprise Architecture and the SA for a specific project.

Exhibit 2-6: Solution Architecture builds from DOI EA Strategy and Planning



2.2 Solution Architecture Principles

Principles establish the basis for a set of rules and behaviors for an organization. There are principles that govern the IEA process and principles that govern the implementation of the DOI Target Solution Reference Architecture (TSRA). Architectural principles for the IEA process affect development, maintenance, and use of the IEA. These principles are documented in the DOI Conceptual Architecture Document [17]. TSRA principles for implementation establish the first tenets and related decision-making guidance for designing and developing information systems. The Solution and Application Architectures will encompass the following principles:

- **Flexibility**
  The DOI will implement service-based application software that is independent of hardware platforms, that is loosely coupled to infrastructure services, that places reasonable demands on networks used for communication (i.e., minimizes traffic), and that places minimal demands on users of the application.
- **Efficiency**
  The DOI will create designs that reduce cost and implementation time, minimize human intervention to preserve continuous and reliable operation, and reduce user-training requirements. The architecture promotes user interfaces that optimize the nature, efficiency, and effectiveness of the human operator.
- **Usability**
  The DOI will implement easy-to-use solutions, accessible by people with dTSRAbilities, which solve DOI business needs and provide needed information to the public.

- **Balance**
  [redacted]
- **Reuse**
  The TSRA, by creating standard application structure and leveraging SOA, provides the foundation for the reuse of assets, including the solution architecture; business, data, application, and run-time patterns; and business and infrastructure services. Reuse of assets is key in reducing cost by eliminating redundant systems and promoting best practices across the department.

2.3 Constituent Architecture Domains

The goal of Solution Architecture (SA) is to structure every application so that it is cost effective, usable, and maintainable, and so that it meets both its functional and non-functional requirements. Enterprise Architecture, on the other hand, aims to achieve standardization, interoperability and sharing across all the business processes and IT applications within the enterprise. These are different, but complementary, goals. Hence, the evolution of the two architectures must be coordinated.

Consider this example. Each bureau within DOI builds applications that fulfill the respective business needs. If, overtime, we discover that many applications provide web interfaces to existing systems, it would be more sensible to structure the applications in a like manner from the start. This would reduce complexity and increase the opportunity for software and hardware reuse. Additionally, if all our applications have a common requirement to send paper mails, we would prefer a common mechanism for printing and mailing. Furthermore, there will eventually be numerous applications within the enterprise that we would like to interoperate and share information. The most effective way to achieve these is to align all solution architectures to the overarching enterprise architecture.

The IEA has defined its reference architectures and the Methodology for Business Transformation (MBT) to identify DOI's business needs and create plans to address them, including what solutions should be implemented. The TSRA ensures these solutions are aligned to the enterprise architecture. As shown in Exhibit 2-2, SA encompasses elements of the Business, Data, Application, Technology, [redacted] The interaction of these architectures within SA is discussed in the next five subsections. For more information on the IEA, see http://www.doi.gov/ocio/architecture/fea.htm.

2.3.1 Business Architecture

Solution Architecture uses Business Architecture to ensure that DOI business needs are addressed correctly and completely by every solution. Business Architecture, as defined in the IEA, identifies DOI business domains, practices and policies and is developed by using the MBT to create a Modernization Blueprint.. Each solution will address a particular need of one or more Lines of Business. Business Architecture also helps determine the appropriate business patterns for a solution, which in turn affect the choice of application patterns. For more information about the DOI BRM, see http://www.doi.gov/ocio/architecture/fea.htm#brm.

2.3.2 Data Architecture

Solution Architecture uses Data Architecture to ensure that an application's data is complete, standardized, and can be accessed and reused across the enterprise (DOI). To this end, data in each DOI application should conform or map to DOI DRM Entities and Attributes. While this may not be feasible in large-scale COTS implementations, the mapping between main information classes must be done and mapping to important entities should be considered. For more information about the DOI DRM, see http://www.doi.gov/ocio/architecture/fea.htm#drm.

2.3.3 Application Architecture

The DOI Application Architecture and associated technology-specific Application Reference Architectures are described in Chapter 5. Application Architecture is not explicitly referenced in the IEA except to the extent to which it is dealt with in the DOI Technology Reference Model (TRM). Most application architecture depends on specific technology solutions. The Software Development Life Cycle (SDLC) documentation will provide the right best-practice artifacts to build DOI solutions.

2.3.4 Technology Architecture

Solution Architecture uses Technology Architecture for guidance in identifying the best platform and infrastructure for each application. Technology Architecture describes the infrastructure on which application components depend for execution, best practices for organizing the infrastructure components, and recommended technologies to be used. As such, it uses the IEA Technology Reference Model (TRM). The TRM contains recommended Technology Specification as well as lists of preferred COTS products for particular types of solutions. For more information about the DOI TRM, see http://www.doi.gov/ocio/architecture/fea.htm#trm.

2.3.5 Security Architecture

[redacted]

2.4 Architectural Styles and Patterns

DOI has a unique mission, different from other agencies and businesses, but common business patterns and solution architecture (SA) based on industry best practices apply to the underlying DOI IT solutions. For example, the architecture for an e-commerce application at DOI will probably resemble an e-commerce application at another agency of like size. This implies that there is a difference between a specific architecture, and the type of application it defines. The common industry term for the latter is *architectural style*.

We define architectural style as "A set of principles, elements, patterns, and constraints designed to meet a specific set of requirements within a specific scope" In other words, an architectural style contains a well-defined set of patterns that constitute a common way for computer system components to interact with one another. For example client/server, standalone, n-tier, and enterprise application integration (EAI) are all examples of architectural styles.

Many metaphors used to explain software architecture are based on construction or building architecture. We use a cathedral as a metaphor to illustrate architectural styles. All cathedrals have certain underlying construction principles; for example the basic floor plan is that of a cross. And while there is also wide variation among cathedrals, a few common characteristics emerge such as Romanesque and Gothic. These two architectural styles define a specific set of patterns that transform the basic cathedral into an easily identifiable style. The choice of an architectural style for enterprise applications is typically made as a result of engineering tradeoffs in response to a specific set of requirements, rather than esthetics. For example, if your application must support geographically remote users, then the standalone architectural style is clearly unsuitable.

At the DOI, the target Solution and Application Architectures use a combination of n-tier and enterprise application integration (EAI) architectural styles. In addition to these broad architectural styles, we use specific business, integration, application, run-time, and design patterns to describe common problems solved by DOI applications. Many applications have the same basic building blocks, such as scanning, printing, and metadata management. It is incumbent on the application architecture to provide standard or common solutions to these common problems that can be shared across applications. This will both maximize the opportunities for reuse and minimize overall maintenance.

The most common use of patterns has been to use 'design patterns' to describe a particular implementation solution. This use is based on the groundbreaking work of Gamma, *et. al*. in the book Design Patterns. In the TSRA, we are using patterns at an architectural level, rather than an implementation level. In his article "Inside Patterns", Douglas Schmidt describes architectural patterns as follows:

> A *pattern for software architecture* describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution. The solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate. [1]

He goes on to say that every pattern:

- documents existing, well-proven design experience,
- identifies and specifies abstractions that are above the level of single classes and instances, or of components,
- provides a common vocabulary and understanding for design principles,
- is a means of documenting software architectures,
- supports the construction of software with defined properties,
- helps with building complex and heterogeneous software architectures, and
- helps to manage software complexity

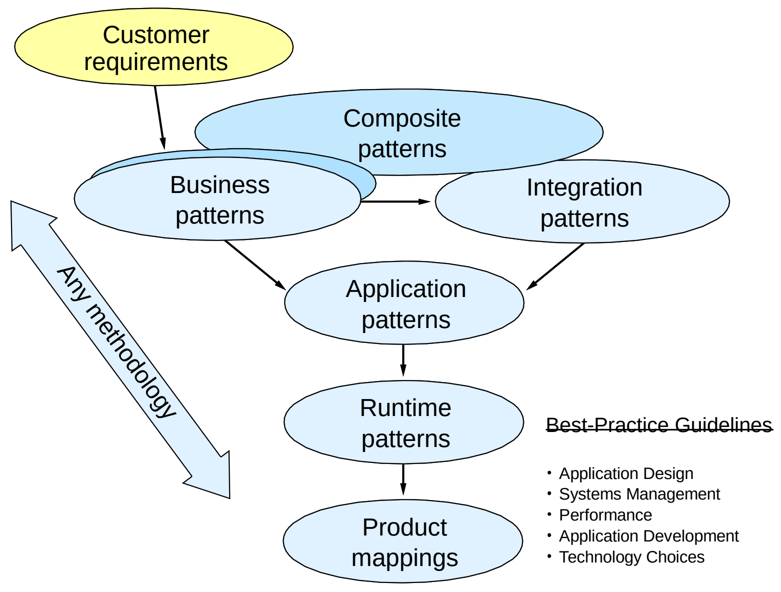
The SA builds on business patterns to provide the framework of the foundation infrastructure, to build its application architecture, and to integrate with the architectural styles in use. The TSRA uses four kinds of patterns, based on “Patterns for e-business: A Strategy for Reuse” [1]:

- Business Patterns
- Integration Patterns
- Application Patterns
- Run-time Patterns

Business and integration patterns reflect the requirements, and are independent of application and infrastructure topologies. Business and integration patterns are sufficiently general that they can be used for both the current and planned architectures. On the other hand, the application and runtime patterns put a stake in the ground that affects application and infrastructure design.

Applying patterns for e-business (selecting which patterns fit best) occurs in steps that parallel the layered asset model shown in Exhibit 2-7:

Exhibit 2-7: Layered Asset Model



Pattern selection begins with identifying the business patterns that apply to the business processes being considered. Simple implementations may involve only one business pattern, but it is more common to find that several business patterns apply. Depending on the nature of the business requirements, integration patterns may also be needed to extend and support the business patterns.

Composite patterns represent frequently occurring combinations of business and integration patterns. A less frequently occurring combination of business and integration patterns constitutes a custom design.

Business and integration patterns correspond to sets of application patterns that provide application topology choices, each optimized for a specific set of business and IT drivers. They are described in Chapter 3, Business Architecture. Application patterns support runtime patterns that provide infrastructure topology alternatives. Finally, products can be mapped to each node of the runtime pattern, using the tested product mappings from DOI's preferred products list.

2.5 Service-Oriented Architecture Overview

Within the TSRA, Service-Oriented Architecture (SOA) is the preferred method for providing business and infrastructure logic. The FEA reference models directly support the development of an SOA and the FEA Assessment Framework category "IT Implementation Improvement" looks for agency plans to evolve to an SOA.

SOA is a composite concept that describes a mode of business process implementation (as an orchestrated collection of services) and prescribes a new way of delivering those services though information technology. Thus it is a process organization concept and a system development paradigm. SOA helps create the IT infrastructure with supporting services, tools and processes intended for the construction and combination of services within a scope that extends beyond a single application. This document discusses SOA in Chapter 5, Application Architecture.

2.6 Solution Objectives and Scope

Solution Architecture will give DOI an enhanced ability to:

- Capture, integrate and share information from other sources
- Identify needs (training, resources, etc.)
- Drive Modernization Blueprints from the conceptual to the physical
- Measure performance of programs and their management
- Meet reporting requirements
- Analyze and prioritize LOB-mandated efforts
- Justify requests and expenditures
- Manage citizen-oriented service programs
- Protect natural and cultural resources

Business Drivers:

- Improve Organizational Efficiency
- Reduce the latency of Business Events
- Easy to adapt during organization changes
- Integration across multiple delivery channels
- Unified customer view across Lines of Businesses (LOB)
- Leverage current assets and infrastructure

IT Drivers:

- Minimize total cost of ownership (TCO)
- Simplify skills base
- Simplify Back end application integration
- Minimize enterprise complexity
- Maintainability
- Availability
- Performance
- Scalability
- [redacted]

3 Business Architecture

Business Architecture is applied to a solution by the following steps:

- Write a Business Description of the solution
- Draw the Solution Overview Diagram (SOD)
- Create the Use Case Model(s)
- Define the functional requirements
- Map business, integration and composite patterns to the solution overview

The Business Description must be the first step because the SOD and functional requirements are derived from it.

3.1 Business Description

The business description defines at a very high-level for a solution the DOI lines of business, business functions, the users, and the interactions between users and functions. It can be as simple as a couple of paragraphs. It should define the business functions that the solution will perform along with the DOI lines of business (LOBs) and external users for which the solution will provide services.

[redacted]

3.2 Solution Overview Diagram

The Solution Overview Diagram represents the business description pictorially, showing the users, business functions, and their connections.

[redacted]

3.3 Use Case Models

The high level Use Case Model describes how the proposed solution components satisfy the DOI functional requirements. The model uses graphical symbols using UML notation and text to specify how users in specific roles will use the system (i.e., use cases). The textual descriptions describing the use cases are from a user's point of view; they do not describe how the system works internally or its internal structure or mechanisms. The Use Case document is *necessary* for the DOI solution to be customized to meet the business's need. Much of the basis for the Use Case Model may be created during the authoring of the Modernization Blueprint. Use Case Model constructs such as Actors, Inputs, Outputs, and Relationships will have been collected during execution of MBT steps 1 - 5 for a given functional area or line of business (LOB)

The Use Case Model is described by the following constructs:

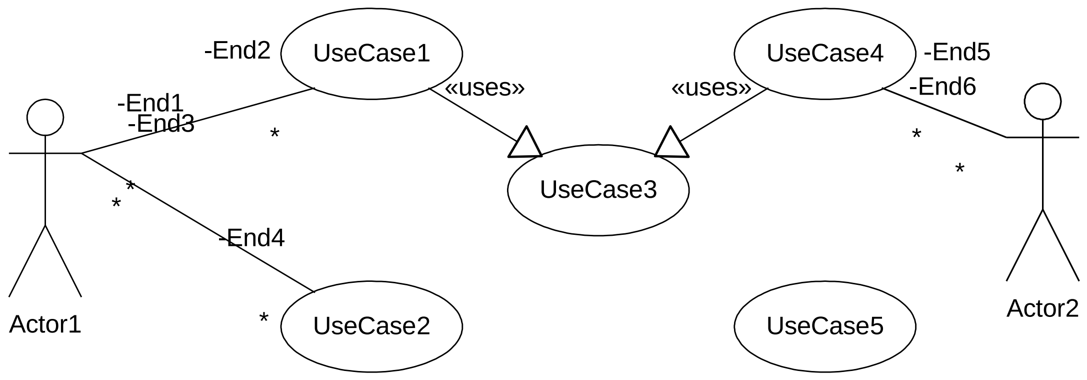
- Actors (name, description, status, subclass, superclass, and associations)
- Use cases (number, subject area, business event, name, overview, preconditions, description, associations, inputs, outputs, traceable to, usability index, and notes)
- Communication-associations between actors and use cases
- Relationships between use cases (same as use case associations)
- Termination outcomes
- Conditions affecting termination outcomes

- Termination outcome decision table
- Use case scenarios (number, termination outcome, description, and notes)
- Problem domain concept definitions
- System steps decision table
- Flow of events table
- System sequence diagram

Actor names, actor descriptions, use case numbers, use case names, and use case business events, and use case overviews as well as communication-associations between the actors and the use cases provides an overview of the functional requirements. The other constructs of the model document the expected usage, user interactions, and behaviors of the system in different styles and depth.

The Actors that participate in the Use Case Model can be graphically depicted to show the relationships among the Actors, and textually defined to describe important properties of the Actors. The following diagram is an example of a high-level Use Case Model.

Exhibit 3-2: Use Case Model Example



Each component in a Use Case Model is defined separately in text documents to describe important properties. The following table is an example defining an actor.

Exhibit 3-3: Example Actor Definition Table

Actor Name	Actor name	
Brief Description	Actor description	
Status	Primary	
Relationships		
Inheritance	**Subclass**	List of actors that are subclasses of this actor
	Superclass	If this actor is a subclass of another actor, name that parent actor
Associations To Use Cases		Use cases this actor is associated with

Use cases are also defined by text that describes details about the Use Case, including the sequence of actions that take place when an Actor carries out a use case.

Exhibit 3-4: Example Use Case Description Template

Use Case Name		
Subject Area		
Business Event		
Actor(s)		
Overview		
Preconditions		
Termination Outcome(s)	**Conditions Affecting Termination Outcome**	**Post Conditions**
Basic Flow	Actor Action	System Response
Alternative Flow 1	Actor Action	System Response
Alternative Flow 2	Actor Action	System Response
Alternative Flow 3	Actor Action	System Response
Input Summary		
Output Summary		
Business Rules		
UC Associations		
Traceability		
Use Case Notes		
System/Database		
Business Volume		
Availability/SLA		
Special Requirements		

Use Case models are applicable to an off-the-shelf system. They are a powerful training tool and an excellent foundation from which to begin customizing an off-the-shelf application. This is their intended role in the DOI project; therefore, high level Use Cases are sufficient and necessary only for the primary functions delivered by off-the-shelf applications.

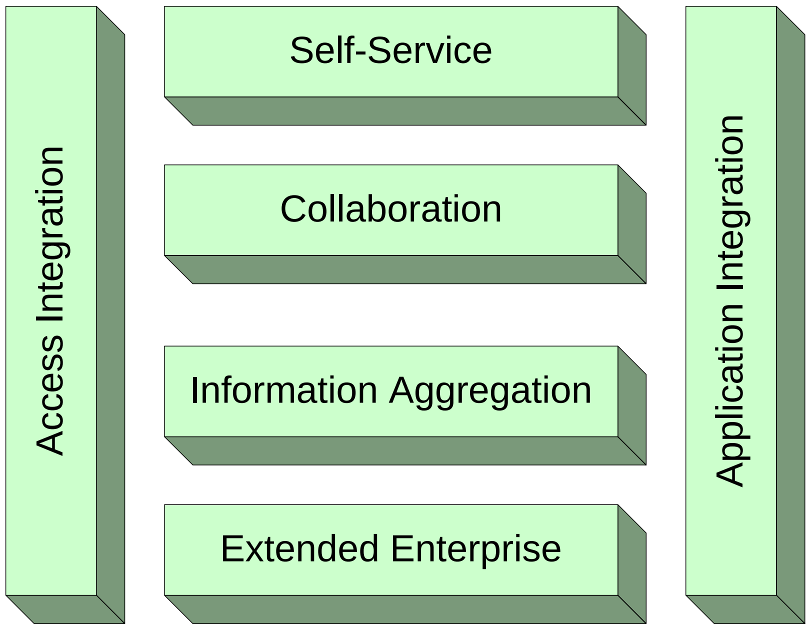
3.4 Functional Requirements Document

A functional requirements document can be useful at a number of points within a system development lifecycle (SDLC), such as the design, testing, and acceptance phases. The functional requirements are captured, primarily from the use case model, normally in a grid presentation, such as a spreadsheet, a table, or a requirement modeling tool. The functional requirements should contain, at a minimum, a statement of the requirement, the date the requirement was captured, its context, source and priority, and the type of requirement (such as business, user, accessibility, etc.). The Functional Requirements Document should tie back to the recommendations in the Modernization Blueprint, if applicable, as well as the solutions overview diagram and the use cases.

3.5 Business, Integration and Composite Patterns

Business patterns are high-level concepts that establish the business purpose of any solution. They define major objectives of the solution, identify participants, and help describe the interactions between participants. Four basic business patterns are at the core of most (if not all) composite business patterns: Self-Service, Collaboration, Information Aggregation, and Extended Enterprise. There are also two integration patterns used to integrate two or more basic business patterns: Access Integration and Application Integration.

Exhibit 3-5: Business and Integration Patterns



These business and integration patterns can be combined as composite patterns to implement DOI-specific business solutions, making up composite patterns, discussed in Section 3.5.3, Composite Patterns. For more information on patterns, see *Patterns for e-business: A Strategy for Reuse* [1] and "Inside Patterns" [10].

3.5.1 Business Patterns

There are four business patterns:

- Self-Service
- Collaboration
- Information Aggregation
- Extended Enterprise

Nearly all DOI business processes involve user interaction with business services, indicating that the self-service pattern applies. For example, the collaboration pattern appears through out the lifecycle of a system development effort where managers, business domain experts and developers work together to define requirements, formulate designs and iterate through the cycles of builds/tests. The information aggregation pattern provides the means to populate the underlying data stores. The need to integrate within DOI and various bureaus and to integrate with business partners represents an example of the extended enterprise pattern.

Exhibit 3-6: Business Pattern Selection Summary

Business Pattern	Description	Applicability to DOI
Self-Service (User-to-Business)	Applications where users interact with a business via the internet or intranet	Good fit for a majority of business sub-processes.
Collaboration (User-to-User)	Applications where technologies support collaborative work between users.	Good fit for a subset of business sub-processes.
Information Aggregation (User-to-Data)	Applications where users can extract useful information from large volumes of data, text, images, etc.	Good fit for a small subset of business sub-processes.
Extended Enterprise (Bus-to-Bus)	Applications linking two or more business processes across separate enterprises	Good fit for the integration with other government agencies.

3.5.1.1 Applying Business Patterns to the SOD

[redacted]

Exhibit 3-7: Solution Overview Diagram with Business Patterns

This step requires an understanding of the business and system context, which should already be documented as described in the repositories mentioned earlier or the Functional Requirement Document (FRD).

3.5.2 Integration Patterns

Integration patterns are the glue that ties business patterns together. There are two integration patterns:

- Access Integration
- Application Integration

Access integration describes recurring designs that enable to access to one or more business patterns. For example, this pattern can be used to create a consistent user interface to common services when accessed from different devices. Application integration allows seamless execution of multiple applications and access to their data to create complex business functions.

Exhibit 3-8: Integration Pattern Selection Summary

Integration Pattern	**Description**	**Applicability to DOI**
Access Integration	Integration of a number of services through a common entry point	Good fit in the area of consistent user interface, single interface to multiple applications in the enterprise

Application Integration	Integration of multiple applications and data sources without the user directly invoking them	Good fit due to the need that there are multiple applications and data stores that a user such as an inspector/ranger needs to interact with for common business processes

3.5.2.1 Applying Integration Patterns to the SOD

Ellipses are added to indicate where Integration patterns apply. Notice in Exhibit 3-9 that two integration patterns have been identified: Application Integration (4 instances) and Access Integration (1 instance).

3.5.3 Composite Patterns

Composite patterns combine business and integration patterns to create complex e-business applications. For example, an enterprise intranet portal aggregates multiple information sources and applications to provide a single, seamless and personalized access to users. For example, the portal composite pattern can potentially use all of the business and integration patterns. Some potentially useful composite patterns for DOI are listed in the following table.

Exhibit 3-10: Table of Composite Patterns

Name	**Description**

Citizen to Government	This pattern is a composite of the self-service, information aggregation, and extended enterprise (for more complicated systems) business patterns with the access integration pattern and possibly the application integration pattern. An application based on this pattern might allow a citizen (the government's "client") to establish and maintain an account with the government site (for example, providing contact information), to view information on the site, and possibly to transact business (for example, applying for a permit).
Government to Government/Business	This pattern (commonly called Business-to-Business or "B2B") might be used for a government "buy-side hub", a site used to facilitate procurement.
Portal	A portal is a secure, personalized point of access to key business information. It typically aggregates information from multiple sources, such as documents, databases, ERP systems, and other line of business applications. Portals present the information through a single integrated, consistent interface that is appropriate for the user's role in the organization.
Enterprise Application Integration	Enterprise Application Integration applies the Access and Application Integration patterns to any number of the four standard business patterns, as implemented in existing applications.
Enterprise Reporting	An enterprise reporting pattern would be used for an application that allows the distribution of reports from diverse applications to anywhere in the enterprise.
Business Intelligence	Business Intelligence is the current buzzword for data warehousing, data mining, knowledge management, and other data analysis applications. It uses a combination of the Information Aggregation business pattern and possibly the Extended Enterprise business pattern with the Application Integration pattern. The general approach is to use "Evaluate, Transform, and Load" (ETL) tools to create data warehouses or data marts, and then use data mining and/or reporting tools to derive valuable information from the aggregated data.
Office Productivity Tools	This pattern encompasses the tools used by information workers, such as email, word processing, spreadsheet, etc.
Knowledge Worker	Knowledge workers are employees who spend a majority of their time analyzing and reporting on information about their business and making decisions based on these analyses. The success of a modern organization depends largely on the ability of its information workers to discover, analyze, and act on line-of-business data and operational information. Knowledge workers most often process information by means of portals and office productivity tools.
Mobile Worker – Mobile	Mobile users rely on notebook computers for their daily work. They are frequently (perhaps generally) disconnected from the internal network and often connect by means of public networks or extranets. To best serve mobile users, applications should leverage Smart Client technology to provide for productive disconnected operation. Connection to back-end systems occurs by means of web services. Smart clients should be capable of connecting over the internet using [redacted] certificate authentication, encryption, and digital signatures. Applications that include free-form text input can also benefit from the inking capabilities of Tablet PCs; this allows data to be entered in handwritten form.
Mobile Worker – Highly Mobile	Highly mobile users require access to business information at all times. They are best served by mobile devices and laptops with wireless connectivity. Extranet web sites can be developed to support mobile devices by optionally producing small-format web pages.
Mobile Worker – Roaming	Roaming users make use of different computers depending on their work location. This is often the case with offices that use a "hoteling" approach for workers who are infrequently on site. Applications written for roaming Windows users can leverage the roaming profile feature of the Windows operating system, and store user data and settings in the user's profile rather than in local computer's file system or registry. The roaming profile feature

	maintains the user's profile on a server and makes it available to the user regardless of which computer is used.
Line of Business Application	Custom-developed line-of-business applications are best suited to specific DOI business activities or the automation of business processes that are specific to DOI. These applications go beyond typical knowledge worker activities and benefit from a more sophisticated user interface that is tailored to the application domain. User interfaces to line-of-business applications use one of two architectures: web applications or rich/smart clients.

4 Data Architecture

The initial steps for a solution relating to Data Architecture are to define the subject areas and information classes used by the solution, to map solution data in the to DOI Conceptual and Logical Entities, and to create a solution Entity Relationship Diagram (ERD) and/or an Object Role Model (ORM) diagram. Data patterns can also be useful when a new system must be developed. Much of the data work is started during the creation of the Modernization Blueprint. The data architecture for a business area is started during Steps 2-4 in the MBT. By the end of the Modernization Blueprint study, the data architecture has been developed to include a target logical data model, an information exchange matrix, a CRUD matrix, and recommendations for authoritative data sources.

At some point in the solution lifecycle, the architects and designers may determine that it makes sense to use an existing database, a COTS database, or a COTS application with its own database structure. If this is the case, then not all data-related steps documented here may be applicable to the solution. For example, it may not be particularly useful to map a COTS database to conceptual and logical entities within the DOI DRM, but it should be rather simple to define the subject areas and information classes within the COTS database.

4.1 Subject Areas and Information Classes

Subject Areas and Information Classes are part of the DOI Data Reference Model (DRM). Subject Areas are collections of data classifications representing broad categories of information that support a line of business. An Information Class is a logical grouping of entities within a Subject Area. The DOI DRM contains 21 subject areas, each having one or more information classes.

[redacted]

4.2 Conceptual and Logical Entities

The conceptual and logical entities take the data classifications down another level to identify "objects" that could map to a (logical or physical) database table. For example, [redacted]

The solution architects and designers should expand the list of subject areas and information classes (as discussed in section 4.1, above) by mapping solution data to the DOI Conceptual and

Logical Entity Relationship Model Diagrams, identifying common entities in the solution data. Solution data entities that do not map to any of the entities in the DRM should be noted.

4.3 Entity Relationship Diagrams

If it has been decided that the solution (or parts of it) must be one or more custom-built applications, the types of data that will be required by each application should be organized into logical and physical Entity Relationship Diagrams (ERDs). The logical ERD represents entities and their attributes (and, optionally, categories) as well as the relationships between them. The physical ERD differs from the logical one in that it represents the tables and columns that will actually be implemented in a database.

The basic ERD diagram is made up of rectangles that represent entities (i.e., tables in the physical ERD) or views and lines that represent the relationships between the entities. A logical ERD may also contain a circle sitting on a horizontal line that represents a category. Attributes (columns in the physical ERD) are listed within the entity and view boxes, with primary and foreign keys noted if known. A physical ERD may differ in several ways from its associated logical ERD. Multiple logical entities may be collapsed into one or more tables. For example, it is standard practice to represent every look-up entity (such as state codes, zip codes, etc.) as separate logical entities, but they may be implemented as a single, generic reference table or as a standard pattern of reference tables that includes a values, grouping, and sub-grouping tables.

4.4 Object Role Model Diagrams

For many newer applications, when custom-built applications are required, an Object Role Model (ORM) diagram may be more useful than an ERD. The ORM starts at a much more conceptual level than an ERD, with no attributes and all “facts” stated as relationships between objects. ORM tools allow ORM models to be mapped to relational database schemas. For more information about ORM diagrams, see *Information Modeling and Relational Databases* [9]. There is also a good overview by the same author on the Microsoft Developers Network (MSDN) site.

5 Application Architecture

This chapter describes the DOI Application Architecture. The concept of an application architecture may first be developed by the EA and SA analysts as part of Step 4 in the MBT. If so, the intent is to have a high level applications architecture as part of the recommendations in the Modernization Blueprint.

First, application patterns are described within the context of particular business and integration patterns. The application patterns are used to determine the appropriate application tiers. The 5-layer architecture is mapped to the model as a point of reference. The application architecture is then expanded to cover a multi-channel, technology independent approach. Next, reference models are introduced to provide a foundation for describing the application architecture and the application's integration with technology-specific infrastructure services. (Example mappings to J2EE and .NET are provided.) Patterns for channel styles and server functionality are presented. This chapter also discusses mapping the Service Component Reference Model (SRM) to a solution's business functionality.

5.1 Application Patterns

Application patterns represent the partitioning of the application logic and data together with the styles of interaction between the logic tiers. They describe the shape of applications in terms of the application boundaries, where data resides and how it is accessed, and how users and processes interact. Each of the business and integration patterns described in sections 3.1 and 3.2, above, has associated application patterns. An application pattern needs to be identified for each of the Business and Integration patterns selected. After the application patterns have been selected, the Business and Integration pattern names can be updated to show which application pattern(s) apply to each Business and Integration pattern. This should not require a change to the box or ellipse layout. Only the pattern names are revised, using the "::" notation mentioned earlier. The results of this step for the two Business Patterns (Self Services & Information Aggregation) on the right side of Exhibit 3-9 are shown in Exhibit 5-1 and Exhibit 5-2.

Exhibit 5-1 is the Self Services run time pattern Application Decomposition Pattern. The DOI Target Solution Reference Architecture (TSRA) must also comply with the DOI Enterprise Service Network Topology (ESN initiative) [redacted].

Exhibit 5-2 is the Information Aggregation Multi-Step Application Run Time Pattern for Business Intelligence. There are some variations to the Multi-Step Application Run Time Pattern. DOI may need to adopt additional variations to support Business Requirements in the future. During proposal and procurement, vendors may propose any new technologies to support Business Intelligence Solutions such as Virtual Operational Data Store, Dynamic Query Routing, etc.

Exhibit 5-3 is the Extended Enterprise Managed Public and Private Processes application pattern structures, a system design that handles different business protocols with different business partners and maps long running external transactions to internal business processes and workflow.

The detailed Solution Architecture (SA) for Exhibit 5-2 and Exhibit 5-3 will not be derived until the vendors have been chosen and the detailed business requirements are finalized for the appropriate deployment.

Exhibit 5-1: DOI Business Self-Service Runtime Pattern Application Decomposition Tier

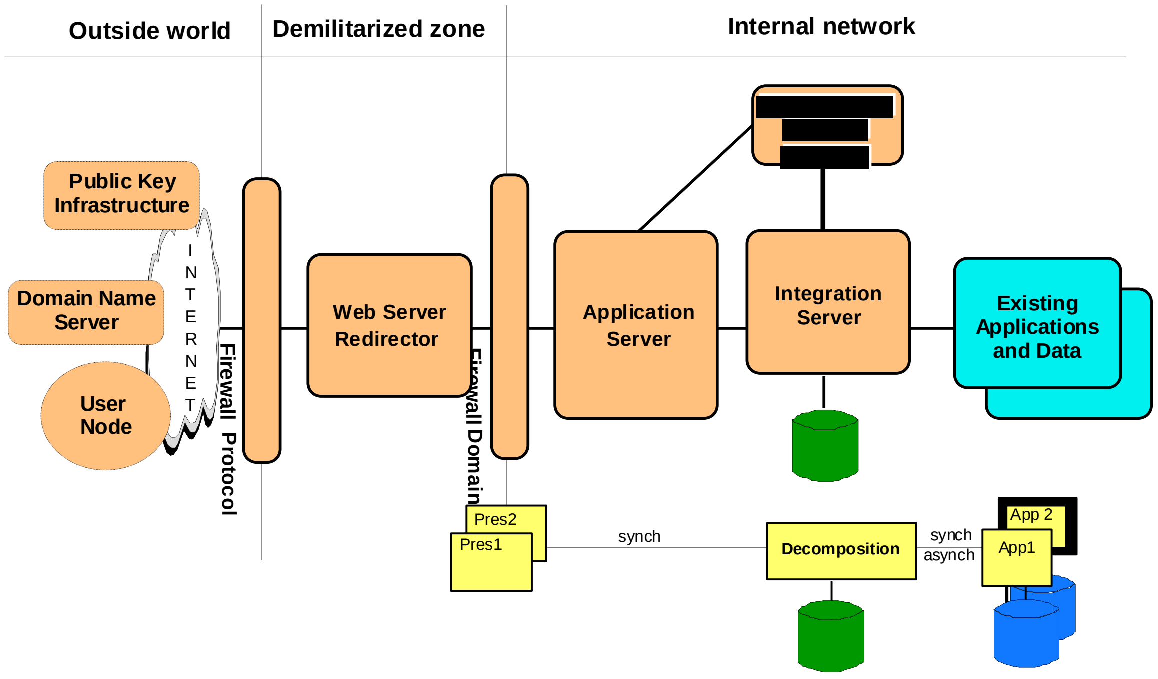


Exhibit 5-2: DOI Information Aggregation Multi-Step Application Runtime Pattern

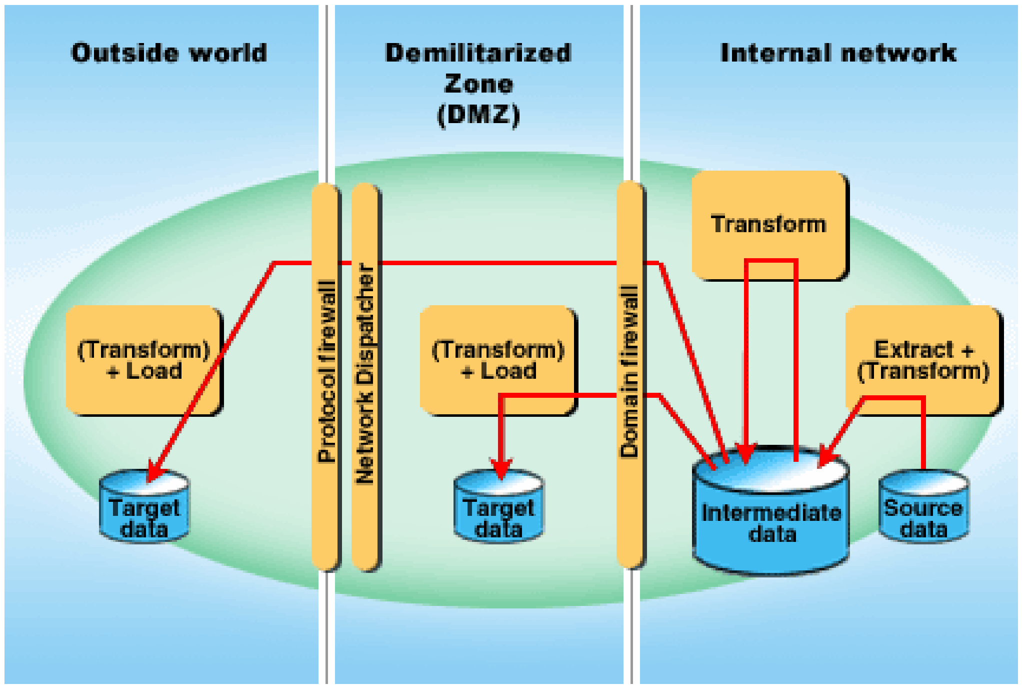
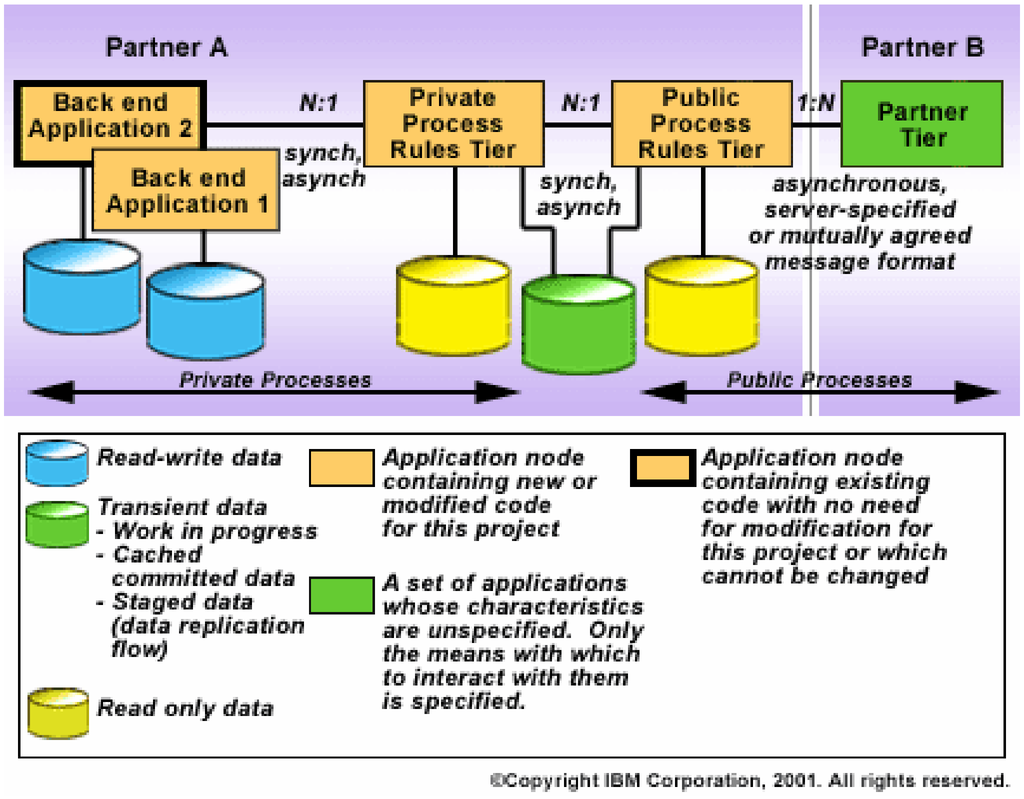


Exhibit 5-3: DOI Extended Enterprise: Manage Public & Private Processes


Partner A
Partner B
Back end Application 2
Back end Application 1
N:1
synch, asynch
Private Process Rules Tier
N:1
synch, asynch
Public Process Rules Tier
1:N
Partner Tier
asynchronous, server-specified or mutually agreed message format
Private Processes
Public Processes
Read-write data
Transient data
- Work in progress
- Cached committed data
- Staged data (data replication flow)
Read only data
Application node containing new or modified code for this project
A set of applications whose characteristics are unspecified. Only the means with which to interact with them is specified.
Application node containing existing code with no need for modification for this project or which cannot be changed
©Copyright IBM Corporation, 2001. All rights reserved.

So far, we have described a generic architecture for constructing distributed applications that meet the DOI requirements. This architecture lays the foundation for all of the DOI Solutions. The next step is to identify the application level functionality that is common across several Solutions. For example several different systems deal with scanning, others deal with application metadata. At a minimum, we want to solve the common problems in the same way in all applications. Even better, would be to create services to perform the common functions and have those services used by the different Solutions.

In addition to representing the partitioning of the application logic and data with interaction styles between tiers, the application patterns are intended to identify the common application functionality across DOI, identify services to provide that functionality, and then provide patterns for how those services are used within applications. DOI has identified eight major application patterns by type of service:

- **Document Management and Scanning** – Covers the full lifecycle of document management from the acquisition of those documents (softcopy text, image, and other formats), to storage, retrieval, and management of the documents. Acquisition of documents includes scanning paper applications, as well as receiving electronic-only applications.
- **Text Search and OCR** - Provides enterprise text search capabilities, text search databases, and creation of text from images.

- **Workflow** – Provides for the definition and management of business processes as workflows. Workflows include both human and AIS performed activities. Workflow includes the scheduling of automated activities, and the tasking and management of human activities using inboxes, outboxes, routing, messaging, etc.
- **Metadata Access and Reporting** - Storing, and retrieving metadata associated with DOI entities, inclusion of the metadata in reports, and the creation of reports about the metadata.
- **Correspondence Generation** - Creation, storage and tracking of outgoing correspondence.
- **Customer Information Management** - Services related to managing metadata relating to external and internal customers.
- [redacted]
- **Application Integration** – Exposing existing applications and/or data as services to the enterprise.

5.2 Non-Functional Requirements

We have already discussed functional requirements. Other requirements, such as availability, capacity, and performance fall into the category of non-functional requirements (NFRs). The Modernization Blueprint and subsequent records of decisions may form the basis for many of the systems functional and non-functional requirements. The blueprint should identify critical performance indicators for a given business area that may be tied to actual performance metrics in Bureau and Departmental strategic plans. NFRs specify the qualitative and other non-functional requirements that an IT system must satisfy. These requirements can pertain to an individual system or a set of systems, to provide a related hierarchy of department- and enterprise-level requirements. NFRs consist of:

- Service level requirements (SLRs), which are run-time properties the system as a whole, or parts of the system, must satisfy. SLRs include:
  - Capacity and performance (volumetrics)
  - Availability
  - [redacted]
  - System management
- SLRs sometimes relate to particular parts of the system, e.g., to particular use cases.
- Other required (non-runtime) properties of the system, such as:
  - portability
  - maintainability.

For convenience, NFRs can also include constraints the system must conform to or satisfy. System Constraints include:

- The business constraints which the system must satisfy (e.g., geographical location)
- The technical standards the system must satisfy

- The technical 'givens' which constrain the system (e.g., which existing hardware or DBMS must be used).

These requirements facilitate the design and development of the operational model (i.e. the computers, networks, and other platforms on which the application will execute and by which it is managed). They also feed into the design of technical and application components. For example, service level requirements may imply component performance requirements. For the implementation of a system using off-the-shelf applications, SLRs define elimination criteria for products that do not conform to the current and target architecture environment in which the system must operate.

It is more convenient to specify the details of certain NFRs in other work products and just refer to them in this work product. For example, use case frequencies could be detailed in the use case model. However, most NFRs should be documented in this section of the SA Document. A subsection is created for each NFR category. The following list can be used as a guideline for the NFR categories that need to be captured:

- Availability
- Backup & Recovery
- Capacity Estimates and Planning
- Configuration Management
- DTSRAster Recovery
- Extendibility/Flexibility
- Failure Management
- Performance
- Reliability
- Scalability
- [redacted]y
- Service Level Agreements
- Standards
- System Management

Some NFRs are expected to change over time. For example, capacity requirements very often increase as more workload is added to the system, or as the volume of stored data increases. Exhibit 5-4 provides a convenient way to capture this information.

Exhibit 5-4: DOI Non-functional Requirement

Capacity Characteristic	**Bureau Level Volume**		**DOI Level Volume**	
	Current	**In 2- 4 yrs**	**Current**	**In 2- 4 yrs**
Average number of concurrent users (including support representatives)	50	500	0	2000
Maximum number of concurrent users (including support reps)	100	1500	0	5000
Total number of users in the system	200	3000	0	10000
Maximum number of unique customer sessions per day. (How many unique	250	2500	0	5000

customers will use system in a day)				
Total number of inquiry transactions processed in a day	50	500	0	1000
Total number of update transactions processed in a day	200	2000	0	4000
Number of transactions that are submitted to external entity in a day	1	250	0	10
Minimum number of inquiry transactions per user session	4	3	4	3
Maximum number of inquiry transactions per user session	8	25	8	25
Minimum number of update transactions per user session	1	1	1	1
Maximum number of update transactions per user session	4	7	4	7

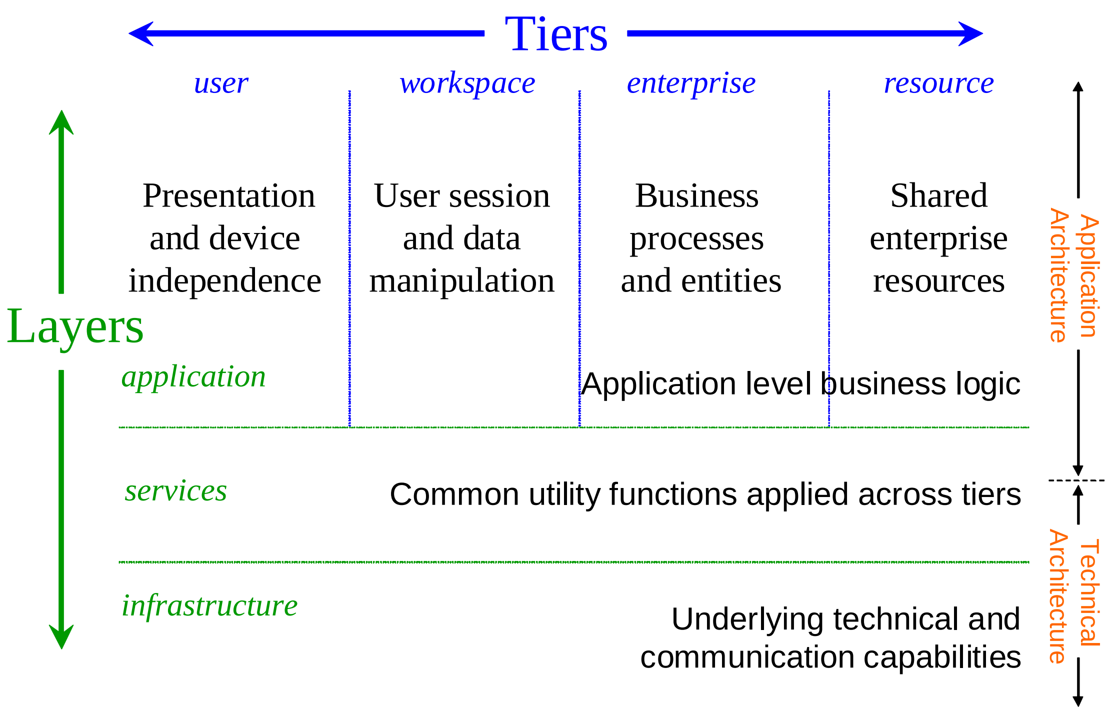
5.3 Architectural Reference Models

The architectural reference model defines the fundamental concepts of the application architecture. The logical reference model is built on two main concepts: ***layers*** and ***tiers***. Both architectural layers and architectural tiers describe a logical separation of functions, where each layer or tier has assigned to it a specific set of roles and responsibilities in response to a specific set of requirements.

The logical separation for architectural layers is chosen based on the need to separate infrastructure capabilities (for example, communications) from technical services (for example, logging or error handling)—and especially from business logic.

The logical separation for architectural tiers—that is, the boundaries between tiers—are chosen or designed to support distribution, scalability, and reuse. Logical tiers can be mapped to different physical computer network topologies. For example, it is entirely possible for all tiers to reside on the same machine. In complex distributed environments, a single logical tier might run on a farm of servers.

Exhibit 5-5: Architectural Reference Model


Tiers
user
workspace
enterprise
resource
Presentation and device independence
User session and data manipulation
Business processes and entities
Shared enterprise resources
Layers
application
Application level business logic
services
Common utility functions applied across tiers
infrastructure
Underlying technical and communication capabilities
Application Architecture
Technical Architecture

5.3.1 Reference Model Concepts

The relationship between architectural tiers and layers in the reference model is shown in Exhibit 5-5, which portrays three architectural layers (at left) and four architectural tiers (at the top).

In terms of Solution Architecture, the application architecture is primarily concerned with the tiers of the application layer. This is where the logical structure of the application is described and the roles and responsibilities are defined. The technical architecture is primarily concerned with the infrastructure layer. This is where the physical realization of the application is defined in terms of software and hardware.

The services layer is addressed by both the application and the technical architectures. For example, the application architecture defines common application level services, such as document management. The technical architecture defines infrastructure level services such as naming/location, [redacted] For each service, there are both application level concerns (how to use the service in an application) and technology level concerns (how the service is implemented). The main distinguishing characteristic of the service layer is that it provides 'common utility functionality' (rather than business functionality) that typically spans the tiers.

5.3.1.1 Layers

Infrastructure is the lowest layer and provides the communications capabilities, among other things. Typically, some or all of the functions in this layer are obtained as part of an off-the-shelf middleware or application server package. The separation of infrastructure into a separate layer protects the application logic from changes in the underlying platform and products.

The Services layer provides utility functions that are useful in more than one tier and by more than one family of applications. Services include capabilities such as logging, configuration, and XML parsing and persistence. Services are independent, callable (frequently remote) functions that provide shared access to resources and capabilities. Some of the services at this layer may be provided directly by the infrastructure, but experience has shown that these out-of-the-box services frequently need to be customized to meet the specific requirements of the enterprise. Independent of any functional customization, the service layer typically provides a higher-level interface that simplifies use of the infrastructure and insulates the application from technology specifics.

Application and business functions are implemented in the Application layer using the capabilities of both the infrastructure and services layers. The application layer is where the roles and responsibilities of the four architectural tiers are exploited.

5.3.1.2 Distribution Tiers

The dotted vertical lines in Exhibit 5-5 represent logical distribution tiers in the model. There are four tiers: user, workspace, enterprise and resource. The distribution tiers are designed to support scalable distribution and deployment flexibility. At deployment time, the distribution tiers are mapped to a specific physical node (for example, the User tier could be mapped to a thin client, the Workspace tier to a web server and the Enterprise and Resource tiers being mapped to an application server, or all tiers could be mapped to the same Unix Workstation system).

Tiers have the responsibility of mediating the flow of data into and out of the system itself. However, each tier has a specific set of roles and responsibilities and the boundaries between the tiers are carefully constructed to achieve the architectural goals. The four-tier model evolved from the classic three-tier architectural model in response to the demands of modern enterprise system. The responsibilities of these tiers are:

- The User Tier is where the system experiences a single use of the system through a specific presentation. The user tier is responsible for device-specific presentation such as that needed for a web browser. The boundary between the user and workspace tier provides for device independence, allowing the application to support multiple devices such as a Web browser and mobile devices, each of which would have its own user tier. The user tier manages user interface details for a single presentation
- The Workspace Tier is where the system supports multiple interactions with a single user. It is responsible for coordinating and maintaining a user session, for manipulating the user data associated with that session, and for interactions with the enterprise tier. The workspace tier:
  - Coordinates and maintains integrity of multiple, concurrent activities for the same user
  - Maintains the user session
  - Provides user preference customization

    - Executes processes that do not require access to enterprise resources
    - Puts and gets data to and from the enterprise

The boundary between the user tier and the workspace tier provides another advantage. It allows the same processing to be used with multiple different devices. In other words, by moving device specifics into a separate tier, we can achieve both reuse of processes, and just as important, consistency of operation across multiple devices. The workspace tier is most commonly implemented with technologies such as ASP .NET or JSP and servlets.

Together, the User and Workspace Tiers support all of the interaction between the system and a single user (or other external partner). There will be an instance of the combined User tier and Workspace tier for each user of the system.

In contrast, the Enterprise and Resource Tiers together provide resources and services to all users of the system. There is only one instance of the enterprise and resource tiers, which are shared by all users:

- The Enterprise Tier is responsible for implementing business processes and entities, and for making their functions available via service-oriented interfaces. The enterprise tier:
  - Maintains the integrity of enterprise resources
  - Enforces system level business rules
  - Provides the scope and control for two-phase commit transactions
  - Provides enterprise services to requestors

  The boundary between the workspace and enterprise tiers provides a clear separation between the resources of the enterprise and the resources required to support a single user. This break allows enterprise resources to be better managed and protected. It also provides a clear access point for all enterprise services, so that they can be shared and reused by multiple applications and users. The enterprise tier is typically implemented with technologies such as .NET business components, or J2EE EJB's.

- The Resource Tier is responsible for the management and access of shared enterprise resources. The resource tier:
  - Provides access to shared resources of the enterprise
  - Provides access to enterprise data and databases
  - Provides access to applications such as COTS or legacy systems

The boundary between the resource tier and the enterprise tier provides a separation between the technology specifics of the resources and the enterprise's use (as well as the service's representation) of them. This allows changes in the resource or enterprise tiers to occur independently, without disruption of the other.

Tier partitioning makes it possible to define specific roles and responsibilities within the system, and to draw clear boundaries among them. Tiers cooperate (that is, invoke each others' services) much the way layers do. However, every tier also invokes the underlying layers to fulfill its responsibilities.

Experience has shown that this four-tier model is most suitable to an enterprise and Service-Oriented Architecture. A modern enterprise must support a wide range of client applications, devices, and access channels and must be able to support flexible, agile, configurable business processes. Each of these requirements is supported by the distribution of responsibility between the four tiers. In addition, this distribution maximizes consistency and reuse of processing logic at multiple points.

In addition to the layers and tiers, the reference model relies on patterns to define the solution to common processing scenarios at the DOI.

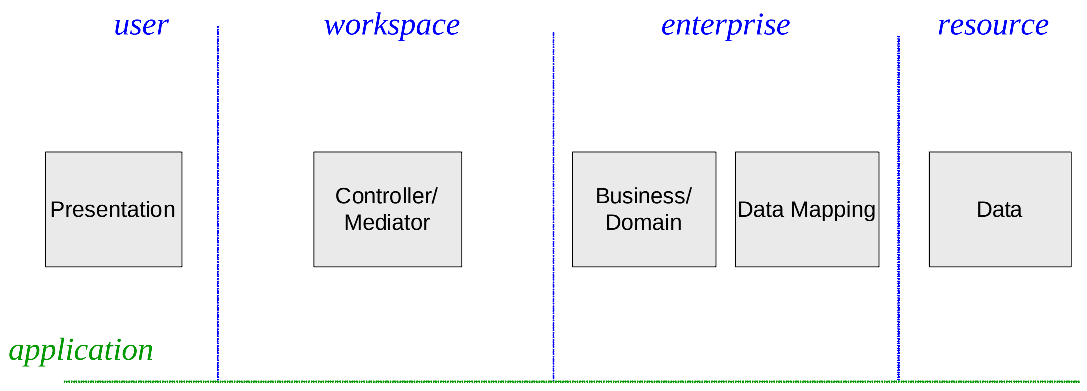
5.3.1.3 Patterns

While these general patterns are appropriate for use within DOI, they are not sufficient. There are also specific application patterns that apply to DOI that are not covered in available literature. The application architecture defines patterns specifically for how to construct common application capabilities.

5.3.2 5-Layer Model

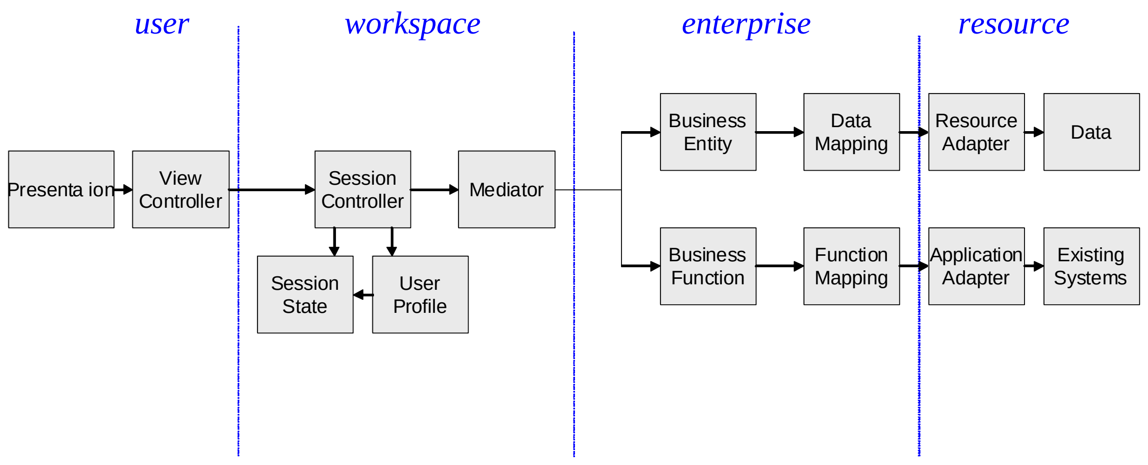
Let’s start to work with the reference model by relating it to another familiar model, namely the 5-Layer model presented earlier. Exhibit 5-6 shows the 5 layers redrawn within the tiers of the new application layer. Their position within the tiers is based on their functions in relation to the roles and responsibilities of the tiers. (Note that we are now using the more formal definition of layers and tiers).

Exhibit 5-6: 5-Layer Model Positioned in the Reference Model



Unfortunately, the model in Exhibit 5-6 still leaves a lot of room for interpretation. What does a controller/mediator do? What is a domain object? How does it relate to an SOA service? The application architecture needs to be more specific if it is to achieve consistency and commonality between applications.

Exhibit 5-7 shows a more detailed version of the 5-Layer architecture that introduces a new concept to the reference architecture. The boxes in the diagram are *architectural elements*, parts of the application logic and structure defined by their roles and responsibilities. The names of the elements suggest their role. These names have been chosen so as not to imply any specific technology implementation. That will occur later in the technology mapping.



5.3.3 Architectural Elements

Each grey box in Exhibit 5-7 represents an Architectural Element in the construction of a software application. These concepts allow the architecture to address specific requirements with specific aspects. The roles, responsibilities and boundaries between elements have been carefully identified to support the following application requirements:

- **Distribution** – the boundaries between tiers correspond to typical machine boundaries in enterprise applications.
- **Scalability** – distribution tiers can be mapped to different physical processors, and each tier within an application can be replicated. This possibility, along with load balancing methods between tiers allows an application to scale up.
- **Separation of application development from infrastructure development** – all software aspects which are not directly related to the business logic of an application are handled in specific infrastructural elements (such as adapters). This separation allows for better reuse of infrastructure, and allows for better resource and skill set utilization of developers.
- **Technology independence** – the architectural concepts and design guidelines are not constrained by a particular technology or product. Each architectural element can be mapped to a different type of implementation (such as a Java servlet or ASP.NET Web Forms), depending on the target technology. The Technical Architecture provides the mapping of the application architecture to technologies and products.
- **Device independence** – Components use resource adapters to isolate business logic from the specific characteristic of storage devices. Presentations are specific to a particular

presentation device and supported by a view controller for that presentation device. Session coordinators are device independent.

- **Application integration** – concepts and specification of components such as data adapter and application adapter are introduced for integrating to the legacy and the packaged application.
- **Future enhancements and migrations** – functional elements may be replaced or split into more detailed elements for any enhancement of the architecture. Separation of concerns allows a designer to replace or enhance "parts" of the architecture with little impact on the rest.

The following presents a detailed description of the different architectural elements shown in Exhibit 5-7.

5.3.3.1 Presentation

We can think of a user interface as being made up of form and function. The presentation element is the form or layout part of the user interface. It is responsible for formatting the input and output of information on behalf of the user. The implementation of a presentation element is dependent on the physical device and technology used for presentation.

For example, an HTML page (perhaps described by a JSP, XForm, or .NET Master Page) could be the presentation for a web browser interface. A Form would be the presentation for a .NET Smart Client interface. A different type of presentation would be used for a handheld device.

5.3.3.2 View Controller

The view controller is the functional part of the user interface. It is responsible for presentation specific input and output, or actually sending data to and receiving data from the presentation device. It is also responsible for trivial business logic such as filtering, drag/drop (where the technology allows it), and cross-field validation. The type of view controller is dependent on the physical device and technology used for the presentation, and normally matches the presentation technology.

For example, if the presentation is HTML, the View Controller could be implemented as a Java Servlet or a .NET class. In Visual Basic (VB), the View Controller is often implemented automatically as part of the VB runtime. If the user interface were through a telephone, then the View Controller could understand about the speech simulation, voice recognition and/or keypad input technologies driven by the Presentation.

5.3.3.3 Session Controller

An application frequently requires more data to be input than can be accomplished in a single presentation. Also, specifically what data needs to be input in subsequent presentations may depend on specific information entered in a previous presentation. The session controller is responsible for the collection and temporary storage of information across multiple presentations and for navigation from one view controller to the next. It handles the "user's unit of work" (as opposed to the system's unit of work).

The Session Controller is responsible for maintaining the user's state throughout the session and for applying the user preferences to the user interface and session logic.

5.3.3.4 Mediator

The Mediator is responsible for making requests to the Enterprise Tier, and for receiving responses and/or data from it. The mediator provides a user/workspace side business abstraction that hides the implementation of the business services and reduces the coupling between the workspace and enterprise tiers. The mediator provides client transparency to naming and lookup services and handles the exceptions from the business services.

5.3.3.5 Business Function

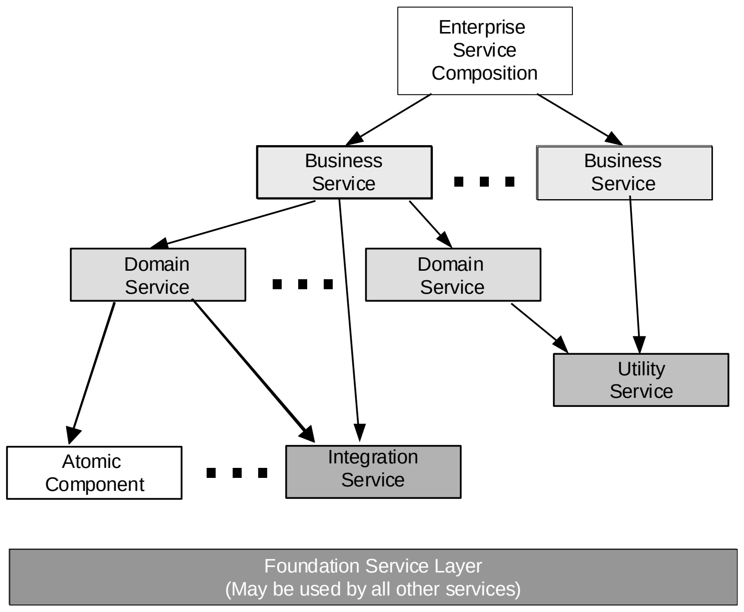
The primary goal of an application is to support functions of the business. The business function (or business application function) element is structured as a service and provides functionality through a well-defined interface, or service contract. Services vary based on granularity, scope, and visibility. The items below define concepts and types of services.

- **Interface** – An interface defines the interaction with a service. It defines the inputs and outputs of the service and any pre-conditions, post-conditions, and constraints of the service. The interface may be specified in terms of an 'interface definition language' (such as WSDL), or as an interface class (as in Java).
- **Granularity** – Refers to the size or amount of functionality in a given interaction. For example, a very fine-grained interaction would be to set or get a single attribute value of an object. A very coarse-grained interaction would be to get all of the values of a collection of objects in a single interaction. These are examples of interface granularity. Granularity also refers to the process and value of a given interaction. For example, 'register trademark' is a higher grained service than 'validate address'. The appropriate granularity of service and interface is based on the intended usage and applies to the type of service.
- **Atomic service** – the lowest level or most fine grained business functionality within the system. Atomic services cannot be decomposed.
- **Foundation Service** – A foundation service is a utility that aids in the construction of business services, such as a business rules engine, data-routing service or workflow system. These services do not provide any specific business functionality, but rather provide higher-level technical capabilities for the construction of services. (Unfortunately, we have been calling these things 'services' for twenty years, so even though they are different than the business /domain services that are now the goals of SOA, we're stuck with the name).
- **Integration Service** – A service that exposes an existing application, legacy or COTS system as a service. It is responsible for mapping between the existing functional and data model, and the enterprise functional and data models.

- **Domain Service** – A domain service is a finer grained service that provides business functionality within a specific business domain. For example, validating correctness of payee data is a service that is shared by several different aspects of application processing. Domain services provide common functionality that is used in the construction of business services.
- **Utility Service** – A lower-level, finer-grained service that provides common, or utility functionality across business domains, such as 'send correspondence'.
- **Business Service** – A business service is a specific kind of service that offers a higher granularity of business value (such as "application acquisition," or "[redacted] It is typically constructed of several lower level or finer-grained processes and services. Business services are often constructed by composing several lower level services together.

The different types of services are used together and support each other. Exhibit 5-8 illustrates the hierarchy of these service types.

Exhibit 5-8: Service Type Hierarchy



5.3.3.6 Business Entity

Business Entities represent units of information (data) that reflect the state of the business. Together with the supporting services (i.e., business function elements) directly associated with the maintenance, access, and provision of that data), they maintain the data and have their own identity. Business entities are often re-used by several different Business Services.

5.3.3.7 Data Mapping

The Data Mapping Entity provides the mapping between the enterprise definition of data (based on the information model) and the physical representation of that data in one or more databases. In many cases, data mapping has to aggregate data from multiple sources to achieve the enterprise wide information view.

Note that we do not attempt to redefine data definitions of existing systems. Attempts at achieving 'enterprise data definitions' failed in the '90s because it is often impossible to change existing systems, much less agree to a common definitions. Instead, we create a new definition of the data needed at the enterprise or business process level, and then map between that new definition and the existing systems.

Object-Relational Data Mapping is used to map data between relational data stores (the typical database today) and objects from an object-oriented design.

5.3.3.8 Function Mapping

As in the case of data, we explicitly call out the mapping between the new services and processes provided in the enterprise tier, and the existing applications that perform some or all of that work. The interfaces of the business services are designed based on the requirements of the business model. Then, those new interfaces are mapped to the existing capabilities of legacy and mainframe systems.

5.3.3.9 Adapters

A goal of the software architecture is to isolate the implementation of business functionality from the specifics of how that functionality or information are implemented or stored. This is achieved through the use of adapters, or special insulating layers. In the case of an existing legacy or packaged application, this is done with an application adapter. In the case of data, this is done with a resource adapter. A common type is an XML-based adapter that transforms a standard XML document into various schemas to allow homogenous consumption.

5.3.3.10 Application Adapter

The application adapter provides a new, modern programming language interface to some existing application functionality. The adapter must then somehow invoke the existing application to perform the work. Typically, some type of protocol and/or communications translation is necessary for this.

The application adapters are implemented as part of the resource tier. They may use some other element, such as a messaging service to gain access to the packaged application. Many application adapters are provided as part of the EAI infrastructure.

5.3.3.11 Resource Adapters

The resource adapter provides access to information resources that exist in one or more parts of the system, such as a database or as part of an existing application. The resource adapter is responsible for providing the data mapper with access to that data. In many instances, the resource adapter will be a standards-based component, such as an ODBC driver. However, when the data resides in legacy applications, the resource adapter may be much more customized.

5.3.3.12 Data

The actual information stored in databases, systems of record, and applications. Within the category of Data as an architectural element, an application must design how data is collected (Data Assembly) and how it is transferred between components and between itself and other applications (Data Transfer).

5.3.3.13 Existing Systems

Existing systems are the legacy, COTS, and other systems and applications that provide existing functionality.

5.3.4 Integration Capabilities

Exhibit 5-7 presented an improved version of the 5-layer model that provided a more detailed and technology independent application architecture. However, there are several additional concepts that we need to introduce to meet all of the requirements of the DOI. Exhibit 5-9 adds the integration concepts of function routing, event mapping and integration services.

5.3.4.1 Function Routing

One of the fundamental tenants of architecture is separation of concerns. There are many different separations, such as the separation of presentation from business logic as promoted by n-tier architectures. Integration technologies have promoted the separation of routing logic from business logic. This allows the routing to be optimized in the infrastructure, separate from the application business components. The routing function architecture element represents this routing logic. It can be mapped to multiple different technologies, such as a content based routing service in an ESB product, or routing specification in an integration broker.

5.3.4.2 Integration Service

Traditionally, EAI technologies have been used for bringing the functionality of legacy and COTS applications into the overall enterprise. The introduction of SOA as a design paradigm requires an addition to the traditional EAI approach. Integration services are a specific use of SOA platform technologies, usually Web Services or ESBs, for implementing the integration of legacy systems.

There are two major types of integration services that are used in the enterprise:

- **Data integration services** – this type of integration service is created to provide data integration between multiple applications. They are usually invoked by the application or legacy system in which data has changed. The data integration service pattern is usually implemented in cases where multiple applications work mostly independently, but data changes in one applications impact the functioning of other applications.
- **Functional integration services** – this type of integration services is created to provide shared functionality between multiple applications. It is invoked by the application, requiring the functionality. The functionality can be complex, or as simple as getting or setting the other application's data (in which case it is still data integration, but implemented through an exposed functional interface).

Integration services share a lot of technologies with business services, but architecturally they are very different:

- Integration services are not exposed directly to the enterprise business processes, but rather to other services which are responsible for wrapping "integration services" in order to expose them to the enterprise. These services serve as a wrapper (implementing the functional or data mapping elements) thus separating the enterprise business services (which are defined based on the business and information architecture) from the APIs and data of the existing applications.
- Unlike business services, integration services don't have to be coarse granularity. The granularity of the "integration service is defined by the granularity of the functionality exposed by the application.
- ACID transactions are often a required property of the integration services, especially when setting data to the application.

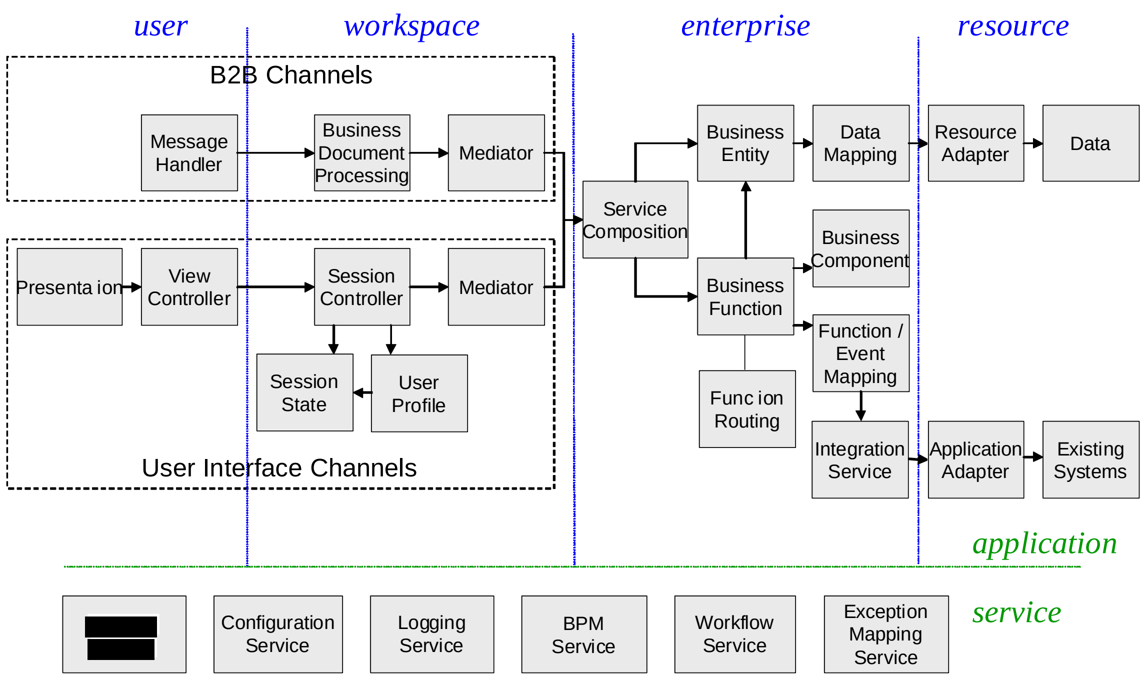
The business service implementation exposes existing legacy system functionality by encapsulating the integration service, which allows for extension of the legacy system functionality without touching existing legacy systems. It also makes it possible to increase the granularity of the service by combining the functionality of multiple legacy systems (or multiple interfaces of the same legacy system) and implementing additional functionality. The integration service implementation also allows for the rationalization of legacy data and alignment with the enterprise data model.

5.3.4.3 Function and Event Mapping

Mapping to existing applications and EAI technologies requires two different approaches. In some scenarios, the integration requires mapping of existing application functions to a new interface semantics (application function mapping). However, other scenarios require mapping between the new integration service and an event driven interaction style such as publish / subscribe within the integration infrastructure. This requires an additional event mapping.

5.3.5 Multi-channel Target Architecture

Exhibit 5-9 adds the concepts of multiple access channels, business documents and, service compositions and defines specific services in the foundation service layer to arrive at the target version of the architecture.



5.3.5.1 Service Composition (business process)

The service composition presents a higher-level business service or business process that is constructed by composing several lower level services together. A business process model should be used to describe the business composition as a sequence of activities and decisions. A business process management system should be used to execute the model and invoke the underlying services. We identify two major compositions that differ in terms of scope and granularity.

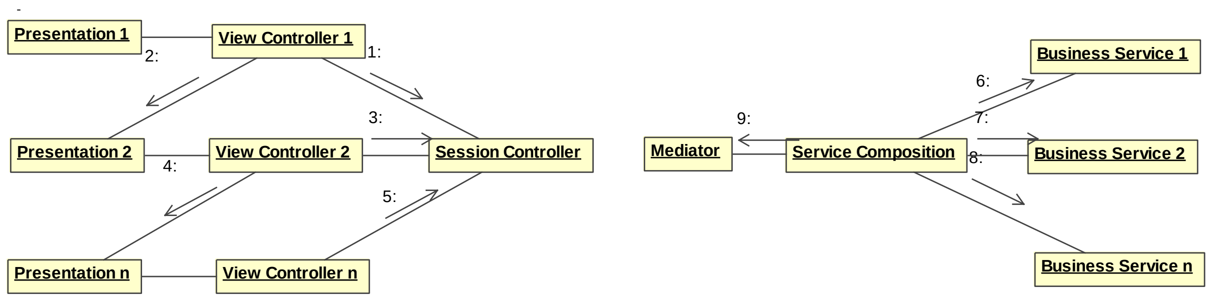
- **Business Process** – A high level process that spans business domains within (or outside) the enterprise.
- **Business Service** – A high level service that provides functionality at the line of business level.

For performance and scalability concerns, it is useful for the enterprise tier to present application functions to any requesters (including the user interface elements) that perform higher level units of work, rather than exporting the smaller units of business functionality to the user-workspace domain. The service composition provides this level of functionality by accepting requests for a given composition, and then invoking, in the appropriate sequence, those services that will execute business functions required to perform the business service.

The interface between mediator and service composition is also frequently the interface between machine boundaries, so a larger granularity interface is required here to achieve well performing distributed applications, particularly in remote client and Telework scenarios.

Once the session (as part of the user workspace domain) has collected a "unit of work" worth of data, it is ready to be passed to the business composition by the mediator. Therefore, the two artifacts are functionally closely related, and are often found in the execution of a single business process. Exhibit 5-10 shows a high level view of this interaction.

Exhibit 5-10: Mediator-Service Composition Interaction



This view clearly expresses the following relationship cardinalities:

- many to one relationship between view controller and session controller
- one to one relationship between mediator and service composition
- one to many relationship between service composition and business services

The session controller cycles through collecting input and navigating to the next page until all of the input screens for a particular use case have been processed. The mediator is then used to request enterprise tier processing of the data. The first step of the business composition may be to start a transaction (if required) and then to execute all of the steps of the process model. When that is finished, the transaction is completed.

5.3.5.2 Message Handler and Business Document Handler

Many systems are not exclusively focused on user interface interaction. DOI has numerous examples of data that comes into the system via files. There are two basic steps in processing this data. First, the message data must be received. This is the responsibility of the message handler element. It may involve receiving a file via an ftp transfer, via a message queue (such as JMS, WebSphereMQ, or MSMQ), or via a web service or some other mechanism. (See Section 11.2.2 for a discussion of various Web Service standards.)

Once the message is received, the ‘business document’ needs to be extracted from the message and transformed to some understood format (note that transformation is not required when the document is received in the required format). This is the responsibility of the business document handler. For example, it may be required to separate multiple applications that are sent in a single file into individual applications. Then, the data needs to be processed and perhaps pre-validated. Again, this is the responsibility of the document handler, which essentially maintains a “session” for the data coming in, and then presents that data to the same enterprise tier interfaces as used by user interface applications. In other words, a document handler is a special kind of controller/mediator for automated or B2B exchanges.

Exhibit 5-9 illustrates this computer-to-computer access style with the “B2B Channel” box at the top of the user/workspace tiers. It also shows the ‘User Interface Channel” at the bottom of the user/workspace tiers. Notice that both of these represent different access mechanisms for getting to the enterprise tier. However, both channels use the same business process in the enterprise tier. This is extremely important in achieving consistent processing, reducing complexity and redundancy and achieving other enterprise goals. The ability to support multiple channels with the same business process is one of the main motivators of the tiers, boundaries, and roles and responsibilities of the application architecture.

5.3.5.3 Service Layer

Foundation Services are common units of functionality, [redacted]. Services can span several different logical tiers, i.e., the same service could be used by elements in the workspace, enterprise, and resource tiers. Services should be implemented in a common manner and conform to an overall service architecture. For DOI we have identified the following services:

- [redacted]
- **Configuration** – Provides a common mechanism for applications and services to get runtime configuration information. This is especially important as more services are introduced because runtime configuration is one of the major customization mechanisms for service.
- **Logging** – Provides a mechanism for writing log and debug information, and correlating the information from multiple distributed services with the same invocation.
- **BPM (Business Process Management)** – Provides a mechanism for composing finer grained business and utility services into higher grained business processes. It also provides transactional coordinating of multiple business services together into an atomic unit of work.
- **Workflow** – Provides a mechanism for managing and organizing the flow of tasks in a business activity. Task may be assigned to individual workers or systems. The Workflow service manages the assignment and flow of tasks from one worker to the next within a given activity, and manages the queue of tasks for each worker.
- **Exception Mapping** – Maps individual AIS exceptions to common DOI exception codes.

5.4 Application Interface Types

One of the main requirements for the application architecture is to support multiple different client access or GUI styles with common processing in the enterprise tier. The client channels are implemented in the user and workspace tiers. The following table (Exhibit 5-11) lists the standard application interface types.

Exhibit 5-11: Application Interface Types

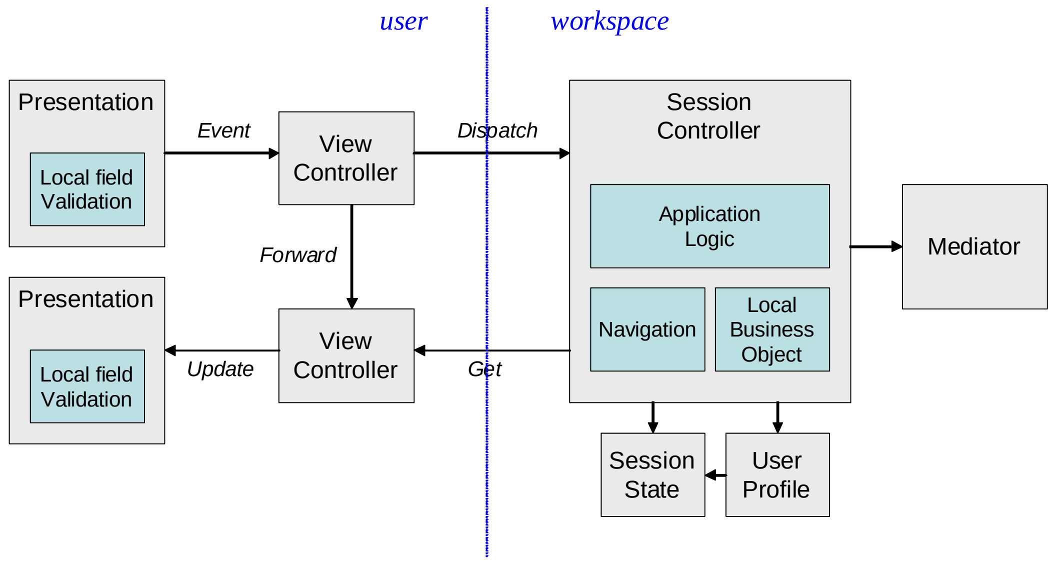
Interface Type	**Advantages**	**DTSRAdvantages**
Thin Client	▪ Runs on any type of client system that supports a web browser (including some mobile devices) ▪ Easy to deploy and manage	▪ Cannot provide richness of functionality ▪ No local storage ▪ May have performance issues, with either network and/or client device
Thick (or Fat) Client	▪ Richness of functionality ▪ Can have direct access to enterprise resources or data	▪ Difficult to deploy, manage and update ▪ Non-managed access to enterprise resources ▪ Duplication of logic ▪ Architectural separation
Rich (or Smart) Client	▪ Richness of functionality ▪ Local storage ▪ Can access enterprise resources or data ▪ Automatic update	▪ May require specific client device type and OS ▪ May not externally accessible

Thick client applications are NOT part of the Target Application Architecture for the dTSRAdvantages mentioned above and because they cannot effectively support the enterprise remote access scenarios required at DOI (telework, work at home, hoteling). A migration strategy away from thick client applications is part of the overall telework architecture.

5.4.1 Thin Client

Thin clients are typically the user interface style for public and consumer access to an organization. Thin clients are also common for external business partners. However, thin clients are not necessarily used for private (internal), data intensive applications. Exhibit 5-12 illustrates the typical logic of thin client access. Notice that the client access capabilities are completely implemented in the user and workspace tiers of the application architecture.

Exhibit 5-12: Thin Client Interaction Style

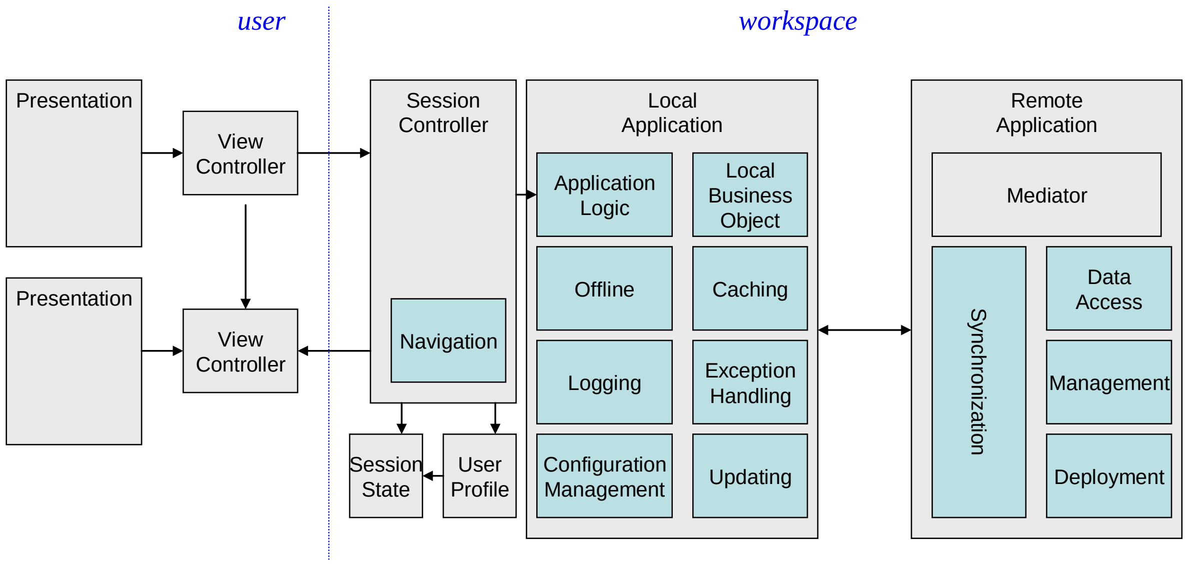


5.4.2 Rich or Smart Client

The rich (or smart) client attempts to offer the best of thick and thin clients by automating the updating and distribution of the application, managing access to server resources, and providing local processing of data and user interface. Of course, it does this at the cost of added complexity. The complexity is managed by the emerging frameworks to support rich clients (Java Server Faces, AJAX and .NET Smart Clients).

Exhibit 5-13 illustrates a typical rich client implementation. Notice that the client process resides in the user and workspace tiers because it operate on behalf of a specific user, not on shared enterprise resources.

Exhibit 5-13: Rich or Smart Client Implementation



5.4.3 Enterprise Remote Access

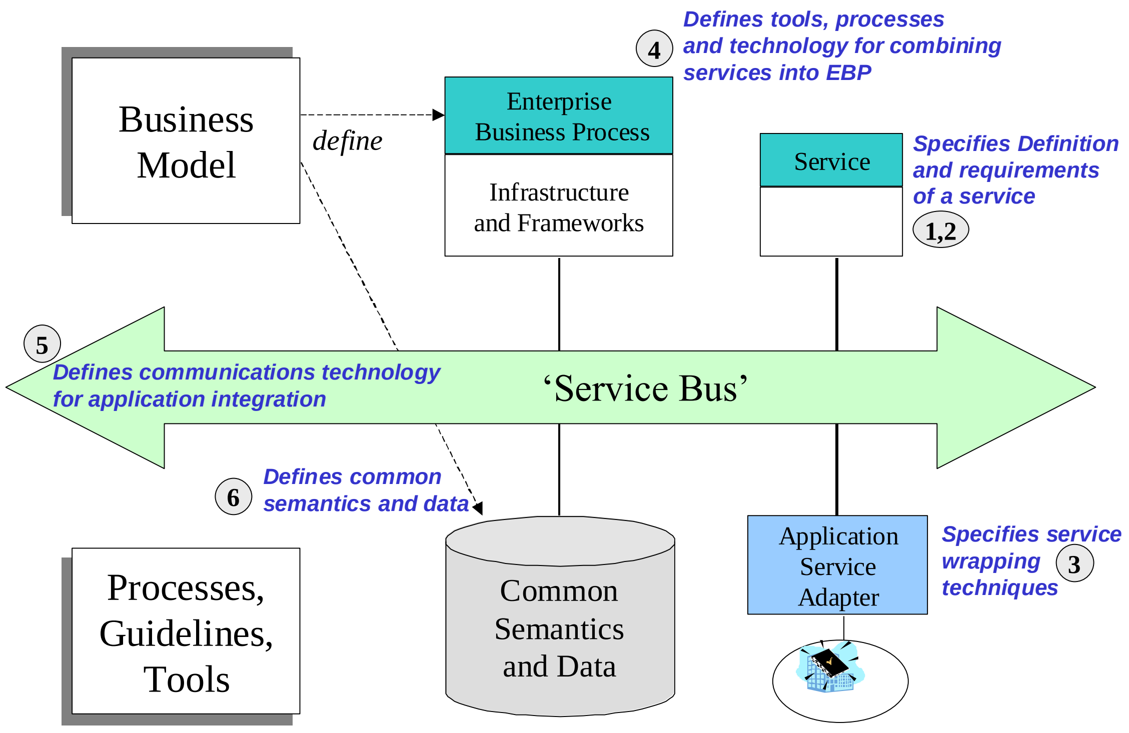
Congress passed the Federal Telework Mandate in 2001. The intent of the mandate was to encourage agencies to provide eligible Federal workers the option of a telework or hoteling environment. The Enterprise Service Network (ESN) group provides remote access to some DOI employees, and is planning for enhanced remote access services for normal telework as well as access in the event of a pandemic.

5.5 Service-Oriented Architecture

Service Oriented Architecture (SOA) is a composite concept describing both a mode of business process implementation (as an orchestrated collection of services) and prescribes a new way of delivering those services though information technology. Thus it is a process organization concept and a system development paradigm. SOA is a distributed architectural style that emphasizes interoperability, reuse, and separation of concerns. A service-oriented system is built from independent service components that discover, interact, and use each other.

At the enterprise level, one of the major goals of an SOA is to build up a library of services that can be combined together into business processes to support and improve enterprise business goals. The use and combination of these services is a focus of business and application patterns described above. In order to achieve this, it is not sufficient to simply build random services, even if they are individually well designed. The SOA must deal with both the construction of services and the combination of them.

Exhibit 5-14: High-level Aspects of SOA


Defines tools, processes and technology for combining services into EBP
4
Business Model
define
Enterprise Business Process
Infrastructure and Frameworks
Service
Specifies Definition and requirements of a service
1,2
5
Defines communications technology for application integration
'Service Bus'
6
Defines common semantics and data
Processes, Guidelines, Tools
Common Semantics and Data
Application Service Adapter
Specifies service wrapping techniques
3

An SOA should describe the following aspects of services:

1. The granularity and types of services;
2. How services are constructed;
3. How existing packaged and legacy systems are integrated into the service environment;
4. How services are combined;
5. How services communicate at a technical level (i.e., how they connect to each other and pass information); and
6. How services interoperate at a semantic level (i.e., how they share common meanings for that information).

Exhibit 5-14 illustrates the various aspects that a service-oriented architecture must address. The numbered circles in the diagram correspond to the numbered list above. Let's look at these in more detail:

- **What is a service?** – An SOA should define the different types and granularities of services, such as domain services, infrastructure services, business services and enterprise business processes. The characteristics (and differences) of each should be clearly specified. (See Section 5.3.3.5, Business Function.)

- **How to use services** – Services are intended to operate within the larger enterprise context (semantic and behavioral environment). The architecture must be clear about how services should be used in an enterprise application (e.g., what standard features they have, what required interactions there are, how they are invoked).
- **How to build services** – The Application Architecture must define, within the SA, the structure of a service and how to build it. For each type of service, the architecture should specify the:
  - Granularity – The appropriate size of the service;
  - Type / style of interface – Guidelines for interface design. For example, business services should be accessed via network interfaces that have different performance requirements than local interfaces have.
  - Configuration mechanisms – Standard mechanisms for configuring services.
  - Other artifacts – The set of artifacts that are required to support a service, such as use cases, component models and specifications, operation models, documentation, test plans, etc.
  - Associated information – Additional information that should be part of a service to support run time and design time inspection, such as version, author, date, keywords, etc.
  - Dependency management and other patterns – Specific design patterns that should be followed to keep services independent and reusable.
- **How to find, evolve and maintain services** – The architecture must describe the complete lifecycle of services, including versioning and backward compatibility requirements.
- **The application infrastructure required to support services** – A service is not valuable in isolation. Rather, its value lies in its ability to be combined with other services to create an agile enterprise. To do this, it must be designed to fit into a specific environment. This environment (infrastructure) and the services it provides must be described by the architecture.
- **The enterprise service bus (ESB)** – In addition to the application infrastructure, the communications infrastructure to enable services to integrate must be specified, along with the guidelines for using that infrastructure. This includes:
  - The communications mechanism – How messages, requests and data are transported. This could be an HTTP connection to a web service (see Section 11.2.2, Web Service Standards) or any other communication protocol.
  - Failover mechanisms – How communication failures are handled, including failover and recovery.
  - Discovery and location transparency – How services are advertised and discovered in a location-transparent manner.
  - Contract negotiation – How service contracts are established between consumer and provider.

- **How to integrate existing applications** into the service environment – The reality is that much of the business functionality at DOI today is not in the form of a service. An essential part of an SOA is how this existing functionality can be exposed as services and connected to the service bus. The SOA must specify the general mechanism for defining these services, wrapping them and connecting them to the bus, with specific implementations for the most common type of system.

- **How to combine services** into larger enterprise business processes – An important goal of an SOA is to enable the reuse of services throughout the enterprise to support a variety of different applications. The SOA must describe the methods, tools and infrastructure for combining services into larger business processes.
- **Common enterprise semantics and data definitions –** The SOA must define the common semantic environment in which the services operate. For example: What data schema must be common throughout DOI for consistency and interoperability? How do [redacted]
- **Business model –** A business model is key to understanding the requirements for a common environment, especially for shared data. The SOA does not necessarily define the business model, but must define how the business model is used to design domain, business and enterprise business processes, and how it drives SOA requirements.
- The **development environment / frameworks / infrastructure / tools** required to support the SOA program – It is not enough to describe what services are; the architecture must enable the easy and efficient creation of those services.
- The **metrics** for measuring program success – An SOA is only effective if it meets the business goals that drive the SOA program. The architecture must choose metrics to demonstrate those goals and a method for collecting and reporting those metrics.

5.5.1 Enterprise Service Bus

The Enterprise Service Bus (ESB) is the infrastructure which underpins a fully integrated and flexible end-to-end service-oriented architecture (SOA)[1]. The ESB enables an SOA by providing the connectivity layer between services. The definition of a service is wide; it is not restricted by a protocol, such as SOAP (Simple Object Access Protocol) or HTTP (Hypertext Transfer Protocol), which connects a service requestor to a service provider; nor does it require that the service be described by a specific standard such as WSDL (Web Services Description Language), though all of these standards are major contributors to the capabilities and progress of the ESB/SOA evolution. A service is a software component that is described by meta-data, which can be understood by a program. The meta-data is published to enable reuse of the service by components that may be remote from it and that need no knowledge of the service implementation beyond its published meta-data. Of course, a well-designed software program may use meta-data to define interfaces between components and may reuse components within the program. The distinguishing feature of a service is that the meta-data descriptions are published to enable reuse of the service in loosely coupled systems, frequently interconnected across networks.

What do we mean by “publishing” a description of a service? Descriptions of the services available from a service provider can be made accessible to developers at the service requestor, possibly through shared development tools. The ESB formalizes this publication by providing a

[1] http://researchweb.watson.ibm.com/journal/sj/444/schmidt html

registry of the services that are available for invocation and the service requestors that will connect to them. The registry is accessible both during development and at runtime. Components such as J2EE** EJBs** (Java** 2 Enterprise Edition Enterprise JavaBeans**) or database-embedded functions may be published as services, but not every J2EE EJB is a service, and not every J2EE EJB is accessible by means of the ESB. In general, EJBs need additional meta-data, and possibly additional bindings, published to the ESB registry in order to make them available as services.

Publication of the service requestors and providers allows their meta-data to be administered through the ESB registry and enables their relationships and interactions to be visualized and updated. Nonetheless, ad hoc requestors and providers may also connect to the ESB without first being registered, for example, subscribers to a "publish/subscribe" topic. In that case, their interactions will not benefit from the full dynamic capabilities of the ESB, described later.

The ESB populates the registry with meta-data about services in three different ways. When services are deployed to the runtime environment, they can be simultaneously and dynamically added to the ESB; meta-data associated with components already deployed can be explicitly added to the ESB; or the ESB can discover services and service interactions that are already deployed and incorporate meta-data describing them in the registry.

Note that the ESB is the infrastructure for interconnecting services, but the term ESB does not include the business logic of the service providers themselves nor the requestor applications, nor does it include the containers that host the services. Hosting containers and free-standing applications are enabled for interaction with ESBs with varying levels of integration, depending on the range of protocols and interoperability standards supported. Most containers (e.g., J2EE application servers, CICS*, Microsoft .NET**) integrate with an ESB across the SOAP/HTTP protocols, but fewer have direct support for SOAP/JMS (Java Messaging Service) over a particular brand of JMS provider. After the ESB has delivered its payload to a container, its responsibilities are fulfilled. Within the container, the service invocation may be redirected among the machines in a cluster, or it may be responded to from a local cache. These are some of the normal optimizations within an application server environment, and they complement the routing and response capabilities of the ESB between the service providers it interconnects. Similarly, the ESB is the connectivity layer for process engines that choreograph the flow of activities between services. The process engine is responsible for ensuring that the correct service capabilities are scheduled in the correct order. It delegates to the ESB the responsibility for delivering the service requests, rerouting them if appropriate.

A core tenet of SOA is that service requestors are independent of the services they invoke. As a result, it is not surprising that the ESB is essentially invisible to the service requestors and providers that use it. A developer can use an API (application programming interface), such as JAX-RPC (Java API for XML-based RPC [remote procedure call]) to a Web service, or distribute messages with the WebSphere* MQI (Message Queue Interface) to a message queue, without considering whether these requests are flowing directly to the service or are traversing an ESB. Similarly, a service provider can be written as a J2EE EJB or a servlet without any specific application code to make it accessible through an ESB. Despite this, one of the values of the ESB is that it takes on the responsibility for many of the infrastructure concerns that might otherwise surface in application code. Thus, although developers can use APIs for service invocation, they do not need to add logic to [redacted].

The ESB virtualizes the services that are made available through the bus. The service requestor, both in its application logic and in its deployment, does not need to have any awareness of the physical realization of the service provider. The requestor does not need to be concerned about the programming language, runtime environment, hardware platform, network address, or current availability of the service provider's implementation. In the ESB, not even a common communication protocol need be shared. The requestor connects to the bus, which takes responsibility for delivering its requests to a service provider, offering the required function and quality of service. Not surprisingly, the infrastructure of the bus is itself virtualized, allowing it to grow or shrink as required by the network and workload which it is supporting.

The flexibility that comes from an SOA, and the virtualization it implies, is fully realized by the dynamic nature of the ESB. All the meta-data, conditions, and constraints used to enable a connection from a requestor to a provider can be discovered, used, and modified at runtime. For example, a new implementation of a service in a different geographical region can be published to the ESB registry, and requests in that region can be routed to it without reconfiguration of the requestors. A service requestor might select a reduced level of assured delivery and see an improved level of performance as the ESB determines that it can use a different delivery protocol. This flexibility is available as a direct consequence of the role of the ESB registry. Because all relevant meta-data for the service provider and service requestors has been placed in the ESB registry, it can be subsequently discovered and used to make dynamic changes.

To achieve much of this flexibility, the ESB accepts requests as messages, then operates on them, or “mediates” them, as they flow through the bus. Mediations can be an integral part of the ESB, providing (for example) transport mapping between SOAP/HTTP and SOAP/JMS, or routing a message to an alternate provider if response times fall below an acceptable value. It is also a feature of the flexibility of the ESB that mediations can be provided by third parties—by other products, ISVs (independent software vendors), or customers—to operate on messages as they flow through the ESB infrastructure. This allows, for example, ISV packages to implement advanced load-balancing features among services, or customers to add auditing to meet new legislation. Mediations can be deployed on the ESB without changing the service requestor or provider.

Mediations are the means by which the ESB can ensure that a service requestor can connect successfully to a service provider. If a service provider requires one format for an address field and a service requestor uses a different one, a mediation can map from one format to another so that the ESB can deliver the service request. If the service provider expects encrypted messages, a mediation can encrypt the “in-the-clear” service requests as they pass through the ESB. The ESB can react dynamically to the requirements of requestors and providers when they are described in their meta-data and held in the registry. In the case of message formats, this is usually achieved through a schema definition. For other service properties, policy statements, which may describe the encryption algorithms to be used or the requirements for auditing, can be associated with the meta-data of the service provider and requestor. The ESB consults this meta-data at runtime and can reconfigure the mediations between requestor and provider to match the requirements. By annotating a policy for the service providers in the ESB registry, the system administrator can, for example, ensure that the services meet the company's new privacy guidelines. Thus the ESB implements an autonomic SOA, reacting to changes in the services it connects.

One of the major uses of mediations is in systems management. Mediations can be deployed in the ESB environment to enable request and response messages to be monitored as they flow through the system, enabling service-level management or problem determination. Mediations can route service invocations to back-up data centers if there is a local problem or to new service providers as they are brought online. They can validate messages in terms of their format correctness, data values, or user authentication and authorization. Through these and other systems management capabilities, the ESB ensures that a loosely coupled and dynamically varying SOA is still manageable in a production environment.2

Many of the mediation capabilities just described are core attributes of the ESB, and the mediations are made available as part of the runtime environment. They are customizable, so that, for example, a generic table-driven routing mediation can be configured to use a specific table and a specific field in a message as the key. The ESB also provides tools to configure the interactions between services—to display the services available in the ESB, to interconnect them, to add policy requirements to a service or group of services, to identify mismatches in the endpoints, and to associate mediations to correct these, either explicitly or through automatic reconciliation of their policy declarations.

Much of the preceding discussion uses the terms service requestor and service provider, as is appropriate for the ESB. Service requestors and service providers are equal partners in the interaction, with the requestor simply being the endpoint that initiated the interaction. The interaction may continue with either endpoint sending or receiving messages. The ESB supports many different types of program interaction: one-way messages as well as requests and responses, asynchronous as well as synchronous invocation, the publish/subscribe model, where multiple responses may be generated for one subscribe request, and complex event processing, where a series of events may be observed or consumed to produce one consequential event. The ESB is also, in principle, transport and protocol "agnostic," with the capability to transform messages to match the requestor's preferred formats to those of the provider. In practice, most ESBs support SOAP/HTTP, which reinforces its role as an interoperability standard. They also support a range of other transports and protocols, some for use by service requestors and providers connected by the ESB, and some for internal communication within the ESB.

5.5.2 Service-Oriented Architecture at DOI

The focus on SOA at DOI is not the construction of a few services, but rather the evolution of the department's enterprise applications to a service-oriented architecture. The SOA requires support from all of the IEA sub-architectures to accomplish this. For example, the business architecture needs to identify process and services as the fundamental building blocks of enterprise functionality. The technology architecture needs to support the publication of location, the invocation, and the communication bus of services. Likewise, there are several requirements for the application architecture to support SOA and the construction of combinable services. The application architecture must define the following service attributes:

- **Granularity** – Not all services are created equal. Services have different granularities (depth of business activities performed) and different scope. A well-defined set of granularities provides a functional hierarchy for building composite applications.
- **Well-defined interfaces and data** – Interfaces should ensure loose coupling and extensibility. Beyond that, services should have similar interaction styles and data definition styles. Services that deal with the same enterprise concept (such as a trademark application) need to use a common definition of that concept in their public interface.
- **Roles, responsibilities, and service grouping** – The application architecture needs to clearly define roles and responsibilities for architectural elements. These roles and responsibilities lead directly to the identification and grouping of services. Service groups help to limit the proliferation of services supporting the same functionality. This depends on the logical focus of the application architecture.

By complying with the TSRA and structuring solutions with service-oriented architecture, applications will achieve the consistency and commonality that will make them cost effective, usable, and maintainable. By integrating solutions across the enterprise, DOI can drive cost down, reduce cycle time, and support a flexible, on-demand business model.

5.6 Component Model

As the process of developing the Solution Architecture (SA) moves from the high-level view found in the Solution Overview Diagram to more detailed views, it becomes useful to create multiple models so that specialized views of the architecture can be depicted. Two important models are the Component Model, which focuses on functional features of the system, and Operational Model, which focuses on the physical runtime infrastructure on which functional components will be deployed.

The value of using multiple models arises from the fact that each of these models begins to call upon different skills and knowledge sets as the level of detail increases. However, since these two models are dependant upon each other, they cannot be created in complete isolation. So, the architecting process now becomes an iterative process of defining portions of each of these two models, then evaluating how each fits with the other, and making revisions that optimize the two models so they support each other effectively.

A Component Model describes the entire hierarchy of functional components, their responsibilities, their (static) relationships, and the way they collaborate to deliver required functionality. A component is a relatively independent part of an IT System and is characterized by its responsibilities, and eventually by the interfaces it offers. Components can be decomposed into smaller components or aggregated into larger components. Some components already exist, but it may be necessary to build or buy others. A component can be a collection of classes, a program (e.g., one that performs event notification), a part of a product, or a hardware device with embedded functional characteristics (e.g., a Personal Digital Assistant (PDA)). Some are primarily concerned with data storage. They can be very large or quite small.

A Component Model evolves through several stages taking into account successive system distribution, the use of specific products, the choice of middleware, and other technologies.

The first level of elaboration (Conceptual) describes the macro level components and a layered architecture that is built on solid architectural principles such as separation of concerns, and emphasizes increasing the cohesion within layers and reducing the coupling that exists between them. This level of elaboration is generally technology-agnostic. This means that it can be implemented using any technology that supports the selected architectural principles (separation of concerns, cohesion within layers & reduces coupling).

The second level of elaboration (Specification) helps to structure and refine the model further by bringing in technology elements such as transport mechanisms, programming models and protocols. Like the Conceptual level, the Specification level is product-agnostic. This means that it can be implemented using any product that supports the selected protocols and programming model.

A third level of elaboration (Physical) realizes the logical components identified in the Specification level of elaboration using specific products and technologies. This model directly maps to a technology-specific application Reference Model. It can be closely tied to the application development tool that is used to implement the system.

The Component Model is described using the following three techniques:

- Component Relationship Diagram
- Component Description
- Component Interaction Diagram

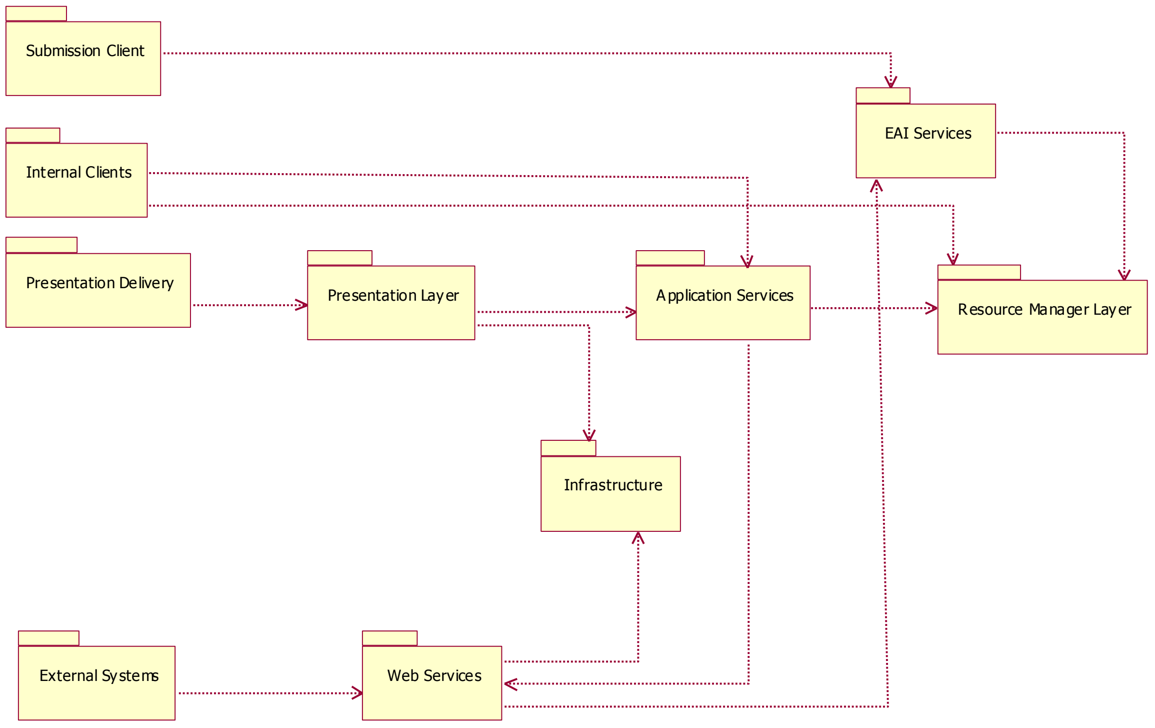
The Component Relationship Diagram and Component Descriptions provide a static view of the model. The Component Interaction Diagram provides a dynamic view of how various components interact when the system responds to an event or request.

Since Patterns for e-business are being used, the Application patterns that have been selected provide a high level tiered topology that can be useful as a starting point for the Component Model. Runtime patterns and product mappings can be used to identify technical components that can be included in the Component Model. <u>Therefore, vendors proposing a solution must provide a high-level component diagram of their proposed product to properly support DOI Business Services.</u>

5.6.1 Component Relationship Diagram

Component Relationship Diagrams can be created using a UML[2] Class Diagram. Use of this notation does not imply that all components must be coded in an object-oriented language. Exhibit 5-15 shows an example of a Conceptual level diagram, with high level components represented as packages.

Exhibit 5-15: High-level Conceptual Component Relationship Diagram

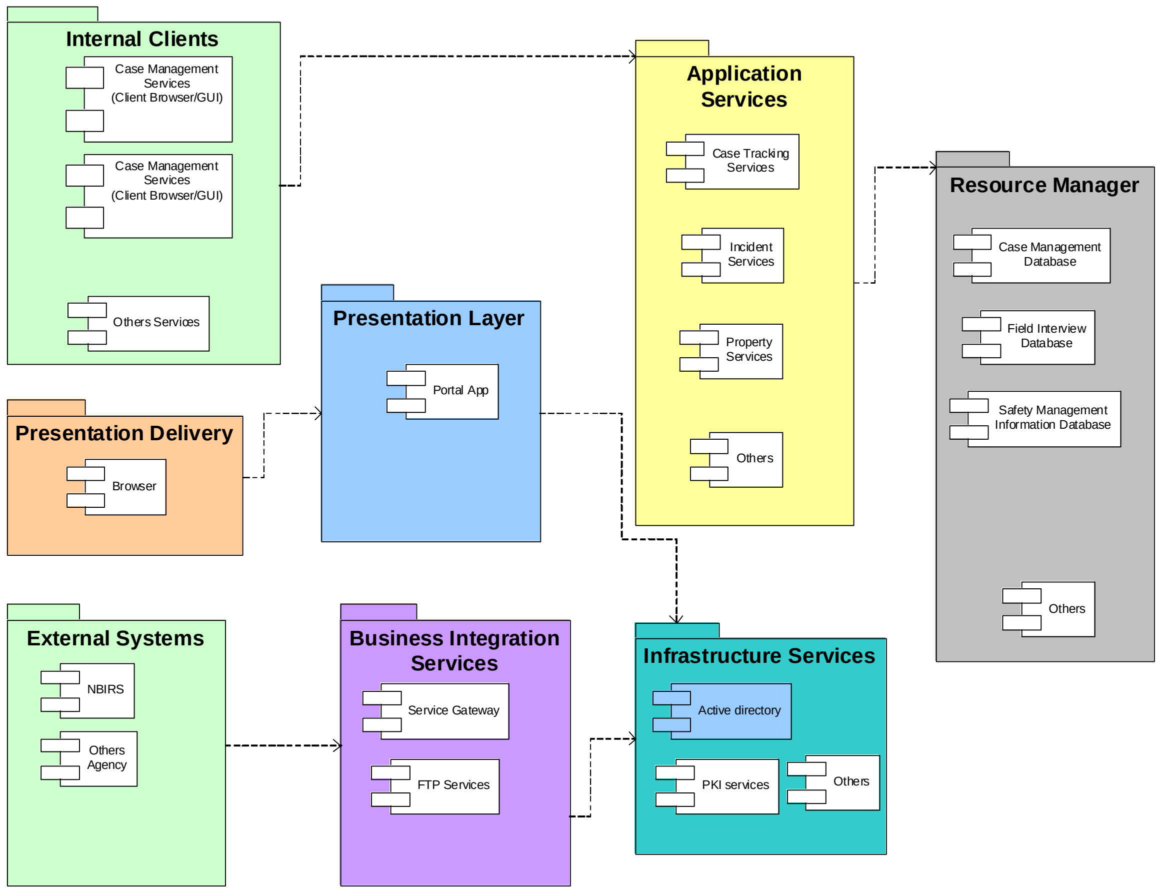


An initial high level component model diagram can be created quickly to show the overall topology of major functional aspects of the system. This view is not yet detailed enough to understand fully what each package will contain, but does allow stakeholders to understand the major features and evaluate the completeness of the architecture.

Exhibit 5-16 shows another Component Relationship Diagram that is the next step in a series of progressive elaborations. In this component model, packages from the first diagram are shown in more detail, clarifying responsibilities that each of the original high-level packages will support.

[2] Unified Modeling Language (UML) is a specification of the Object Management Group (OMG), a not-for-profit computer industry consortium (www.omg.org). For more information on UML, see www.uml.org.

Exhibit 5-16: Example Component Relationship Diagram



5.6.2 Component Description

Each component in the Component Model needs to be described to a level of detail that is directly related to the level of elaboration of its containing model. The amount of detail that is needed in a component description is closely related to who will use the model as input, and what they need to know. The Conceptual and Specification level models are used primarily by the architect as steps toward the Physical level model. The Physical level model is typically used as input to fine-grained design activities, at which point there is often a hand-off from architect to designer, so the Physical level model needs to describe components to the level of detail needed by the designer.

The Conceptual-level component descriptions are typically brief and succinct. At the Specification level, decomposition of the model into more fine grained components typically occurs, and additional detail is often needed to clarify the role of those components. At the physical level of elaboration, greater detail should be supplied so that there is no chance for miscommunication between architect and designer. For projects where DOI/Bureaus plan to implement the initial capability using off-the-shelf applications with minimal customization, the initial Physical level of elaboration should describe the existing capability within the off-the-shelf applications and the integration effort needed to make them work in the existing DOI/Bureau infrastructure environment.

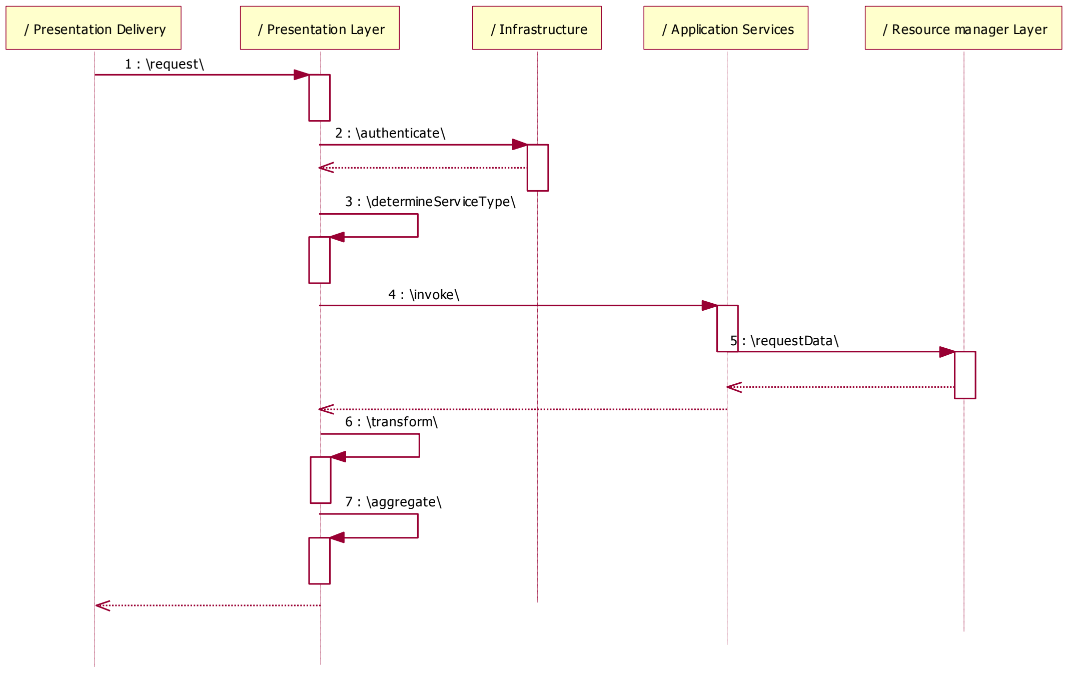
5.6.3 Component Interaction Diagram

All models should provide both static and dynamic descriptions. A model should not be considered complete without both views. The dynamic view of the Component Model can be represented in a Component Interaction Diagram or Activity Diagram (this template shows examples of Component Interaction Diagrams, but Activity Diagrams can be substituted).

A Component Interaction Diagram describes a particular collaboration between components. Component collaborations represent a possible runtime execution. Exchanges occurring between any two components during collaboration are called "interactions."

A Component Interaction Diagram showing collaborations between the top-level components (e.g., Exhibit 5-14, Exhibit 5-15) describe system-wide interactions. Component Interaction Diagrams (associated with more detailed component relationship diagrams) show how the services requested from a component are realized through collaborations among its contained components:

Exhibit 5-17: Example High-level Component Interaction Diagram



5.7 Operational Model

The Operational (or Deployment) Model focuses on the operation of the solution. It is derived primarily from the Non-functional requirements (or operational requirements) that are placed on the application. Like the Component Model, the Operational Model is typically developed through a series of progressively more detailed elaborations (i.e. Conceptual, Specified, and Physical). At each level of elaboration there may be a need to create more than one view of the Operational Model so that no single view becomes overloaded by attempting to convey too much information. By the completion of the physical model, the Operational Model should map directly to one of the DOI Technology-Specific Application Reference Architectures.

A pattern for e-business describes the logical nodes in the Runtime patterns that have been selected can be used as the basis for the operational model.

An Operational model can be used for different purposes at different stages of its development:

- As an early basis for design reviews and walkthroughs, including confirmation that the business problem is well articulated and that there is a viable IT solution.
- As a way of dividing large problems so that each node can be worked on in relative isolation.
- As the basis for early analysis of non-functional requirements such as performance, availability, and capacity, including confirmation of the viability of a solution through specification of the expected non-functional characteristics of nodes and components.
- To identify necessary technical, infrastructure, and other middleware components and subsystems.
- To allow application developers to modify and elaborate their designs based on an early view of how the application will be implemented and managed.
- To contribute to early estimates of the cost of the infrastructure to be used both for budgeting and as part of the business case for the solution.
- As the basis for design reviews and walkthroughs, prior to selecting products.
- As a technical specification against which an architect can evaluate alternative products or even against which technology vendors can submit tenders.
- As the basis for a check that all the necessary business and technical functionality has been identified.

At the Specification level, the Operational Model can be used for the following purposes:

- To document the distribution of application and technical subsystems (deployment units) on preliminary (conceptual or specified) nodes so they can ultimately be installed and run on physical computer systems
- As the basis for detailed design reviews and walkthroughs, prior to selecting products
- As a detailed technical specification against which an architect can evaluate alternative products, or even against which technology vendors can submit tenders
- As the basis for detailed prediction of performance, availability and other service level characteristics (Predictions are based on the overall architecture and the specifications of deployment units within it. They will have to be revisited, via system tests, when specific products have been chosen.)
- As the basis for a check that all the necessary business and technical functionality has been identified

- To allow application developers to refine and confirm their architecture and designs based on a detailed view of all the solution's deployment units
- As the basis for cost estimates of the required infrastructure, using technology neutral costing such as "$ per megabyte of storage"

When all specific infrastructure information has been included in it, the Operational Architecture model is considered to be at the "physical" level, and can be used for the following purposes:

- As a blueprint for the acquisition, installation and subsequent maintenance of the application
- To document how elements at each location are managed, and what extra systems management components and nodes are needed at each location.
- [redacted]

The Operational Model typically includes the following:

- A diagram of the candidate nodes of the architecture and their connectivity. Nodes are potential hardware systems (see Exhibit 5-18 and Exhibit 5-19)
- A description of each candidate node, including its purpose and contained software components (see Exhibit 5-20)
- Several views of the architecture, including a network topology view, an availability/scalability view, [redacted] view, and a view of the development and test environments.
- A walkthrough of a set of business functions. The walkthrough demonstrates the interaction of both nodes and components to accomplish specific tasks.

Exhibit 5-18: Example Operational Model Diagram (J2EE Conceptual Level)

Exhibit 5-19: Example Operational Model Diagram (.NET Conceptual Level)

[redacted]

Exhibit 5-20: Example Node Description Table

Node Name	Node Name
The purpose of this node is …	
Presentation function	
Processing function	
Data	
Infrastructure	
Presentation services	
Processing services	
Data services	
Hardware	
Operating system	
Connections	
Management	
Components contained	
Non-Functional Requirements	
Users and Presentation	
Performance and Capacity	
Availability	
Cost and [redacted]	
[redacted]	

Risk	
Node Management	

The Conceptual Level Operational Model is the basis for additional views that add increasing detail and focus on specific operational aspects of the solution. It is risky to try to capture all operational aspects in a single diagram since the diagram becomes so complex that it is incomprehensible and ineffective. Exhibit 5-18 and Exhibit 5-19 are examples of Operational Model Diagrams for J2EE and .NET at the Conceptual Level.

This Operational Model view identifies specific hardware nodes that will support deployment of components from the Component Model. We can consider this a Physical Level of elaboration since it identifies specific (not conceptual) operational elements of the architecture, the actual number of nodes of various types (chosen and configured to support non-functional requirements such as performance, capacity, availability), and node placement and connection. In this view hardware nodes are emphasized, while network connections are shown in less detail than may be needed to fully understand how to configure the network. Another view can be created that emphasizes network topology details, and de-emphasizing the visual detail of the hardware nodes. <u>Vendors proposing an DOI solution must provide the physical layout of the DOI solution to meet the non functional requirement. In addition, the physical mapping must follow Exhibit 5-18 or Exhibit 5-19 and Exhibit 5-20 of the Operational Model Diagram.</u>

5.7.1 J2EE Application Reference Model

The DOI J2EE Reference Architecture document is the official guidance for the development of custom software based on J2EE and related technologies at the DOI. It describes how to best leverage J2EE technology within DOI and highlights the key architectural topics that have the greatest impact on the success of a J2EE development project. Exhibit 5-18 comes from the J2EE Application Reference Model.

5.7.2 .NET Application Reference Model Overview

The DOI .NET Reference Architecture document is the official guidance for the development of custom software based on Microsoft .NET and related technologies at the DOI. It describes how to best leverage .NET technology within DOI and highlights the key architectural topics that have the greatest impact on the success of a .NET development project. Exhibit 5-19 comes from the .NET Application Reference Model.

5.8 Architectural Decisions

This section describes and captures the rationale and justification for key decisions that affect the architecture. This capability in the SA process provides crucial traceability for the decisions that affect the architecture of a system. Architectural Decisions are most effectively documented when they are organized by subject area. A heading should be placed before each decision so that it can be included in the table of contents.

Decisions should be given a short name and assigned a date and status that indicates if the decision statement is draft, or approved. If a decision needs to be revised, that change should also be noted, along with the date of the decision. Each decision should also be assigned a unique number, and the number system can include an abbreviation for the subject area to simplify maintaining the numbering scheme.

The following table can be used to document architectural decisions:

Exhibit 5-21: Architecture Decision Table Form

Subject Area	**Topic**	**Status**	
(choose one) AD - AD Tiers AM - Application Models DD - Data Distribution DV - Distribution Variations IF – Interfaces TC - Technology Components Other	(short name for this decision)	Draft/Approved/ Revised 2002-mm-dd (author initials)	
Architectural Decision	A concise statement of the decision. (note: the id to the left is a combination of two letter abbreviation for the subject area, and a sequential number. These should not be renumbered)	**AD ID**	xxNNN
Problem Statement	A description of the problem this decision is related to.		
Assumptions	Any pre-existing assumptions or constraints		
Motivation	What is behind the decision thought process, the key principals that apply and what weighting of principals or qualities is being applied.		
Alternative 1			
Alternative n			
Decision	Short description of decision (i.e. which alternative was selected – the alternatives should have explained themselves already)		
Justification	Primary reason(s) why the alternative was decided upon		
Implications	What this means for the system, directly or indirectly		
Derived requirements	If this decision adds some implicit new requirements, what are they?		
Related Decisions	A list of any decisions that are closely related to this decisions		

5.9 Test-Driven Development

Test-driven development is a proven technique (often associated with "Agile Programming"[3]) for ensuring quality deliverables. The approach uses a testing tool such as JUnit (for J2EE) or NUnit (for .NET) to create test cases that can be run again and again.

[3] For more information on Agile Programming, see www.agilealliance.org. Note that Agile Programming is often associated with another concept, eXtreme Programming. Extreme programming practices are typically not acceptable for government development projects.

A test case is created for each requirement and/or condition in the software. Each test case executes one or more components with set test data and determines whether the test was successful. As components are developed, all test cases for each component are run until all execute successfully. Before software is checked into a Configuration Management tool, all test cases for all components should be run to ensure that the added or changed software did not break any other components. Normally, running all test cases should be as simple as a single click with the results displayed in a dashboard that immediately alerts one to any failed tests. The test cases should also be executed before deploying a build of the components.

5.10 Mapping the SRM to Solution Functionality

According to the FEA, "the Service Component Reference Model (SRM) is a business and performance-driven, functional framework that classifies Service Components with respect to how they support business and/or performance objectives." It is intended for use supporting the discovery of business and application service components to provide a foundation to support reuse of applications, application capabilities, components, and business services.

Mapping the appropriate SRM service components to high-level components in a Solution Overview Diagram (SOD) provides an initial opportunity to identify related TRM technical service standards for solution components. This process should be repeated at a number of points during the solution lifecycle. For example, opportunities for infrastructure component reuse might only be identified after a solution's larger application components have been mapped to operational models. Many of the technical standards mapped in this manner may not apply to a given solution, but it is important to examine current and preferred assets to determine whether reuse is possible. Exhibit 5-22 shows several service components mapped to the
[redacted]

Exhibit 5-22: SRM Mapped to SOD

[redacted]

Using this example, one can examine the technical service standards related to SRM's Self Service component (Customer Initiated Assistance type within the Customer Services domain), such as Authentication/Single Sign-on, Help Desk, Performance Management, and Software Distribution. Here, for example, Software Distribution most likely does not apply to this example but there may existing solutions within the Authentication/Single Sign-on standard that can be leveraged.

6 Technology Architecture

The Technology Architecture applies to the Solution Architecture by guiding the following decisions:

- On what platform will the solution run?
- What components of the technology-specific Application Reference Models are required for the solution?
- What business functionality and infrastructure components can be reused or purchased as COTS products?

To these ends, DOI Technology Reference Model (TRM) provides guidance on preferred technology specifications. The CTO Council (CTOC) is responsible for developing, managing, and maintaining the TRM. The CTOC is also the Change Control Board for all enterprise business IT solution and infrastructure changes that have impact across the Department and adjudicates conflicts related to technology standards and policy between technical communities within the Department.

The DOI TRM and Service Component Reference Model (SRM) are part of the Interior Enterprise Architecture (IEA) in accordance with OMB's Federal Enterprise Architecture (FEA). All IEA models are contained in the Department Enterprise Architecture Repository (DEAR), available at www.doi.gov/ocio/architecture/dear.htm. In addition, most bureaus have a Bureau Enterprise Architecture Repository (BEAR) and a bureau-level TRM.

A Service Oriented Architecture places demands upon the infrastructure that are different from traditional application architectures. A flexible, virtualized IT infrastructure is required to rapidly respond to on demand needs[4]. A collaboration of several IBM business units has defined the On Demand Operating Environment (ODOE) to provide a set of integration and infrastructure-management capabilities to enable this rapid response. These modularized capabilities can be selected as needed and combined into various solutions to satisfy the needs of a company's on demand business initiatives. The ODOE (see Exhibit 6-1) enables a multidimensional infrastructure framework to facilitate SOA, support pluggable application services and business processes, and create business partner services, choreographed processes, and utility and resource virtualization.

[4] http://researchweb.watson.ibm.com/journal/sj/444/bieberstein html

Exhibit 6-1: On Demand Operating Environment

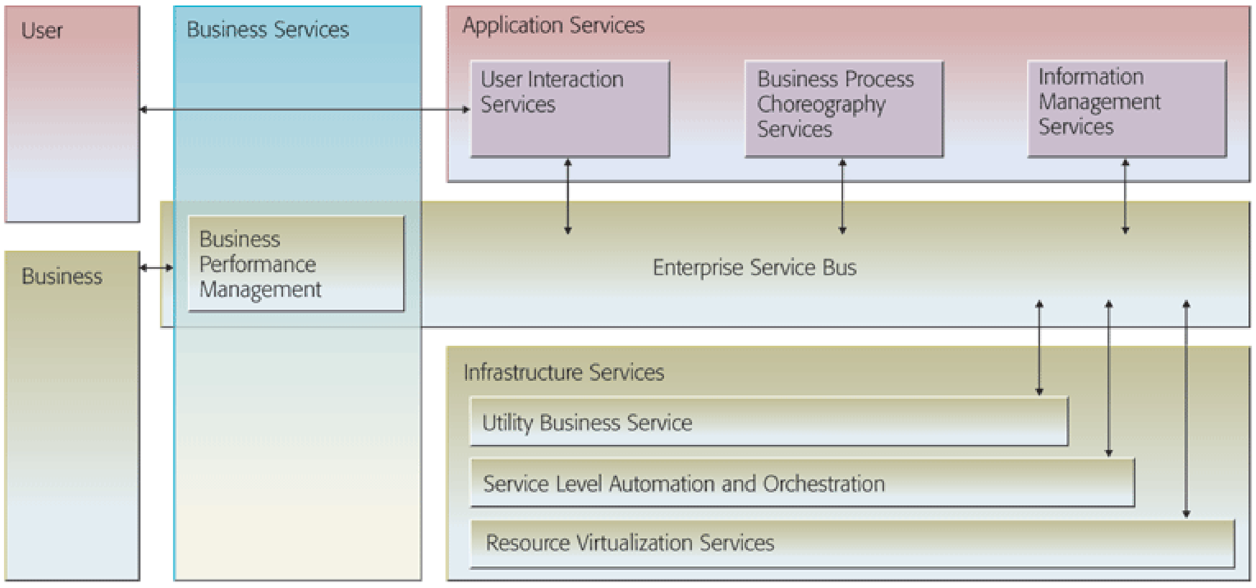

User
Business Services
Application Services
User Interaction Services
Business Process Choreography Services
Information Management Services
Business
Business Performance Management
Enterprise Service Bus
Infrastructure Services
Utility Business Service
Service Level Automation and Orchestration
Resource Virtualization Services

Figure 1
On Demand Operating Environment

The need for an SOA-capable infrastructure that supports the unique requirements of service operational management is often overlooked until after an initial set of services has been deployed. Experience has shown that traditional IT management products cannot support maintenance of service-based systems. Investment in an ODOE is critical for implementing service access management products in conjunction with other operational tools that monitor and manage service responses and requests. Enterprise repositories, such as Universal Description, Discovery, and Integration (UDDI), and approaches based on the Reusable Asset Specification (RAS), provide support for an enterprise-wide, systematic, and regulated pattern of reuse.

[redacted]

[redacted]

[redacted]

7.1 User [redacted] Levels and Procedures

User roles and procedures is the primary focus of [redacted] To implement a solution, [redacted] procedures must be defined that will allow the appropriate users to access and interact with the solution.

[redacted]

particular project data must be appropriately restricted. There may also be a need to define roles for update versus read-only access to groups of data.

7.3 Application [redacted] Policy and Boundaries

[redacted]

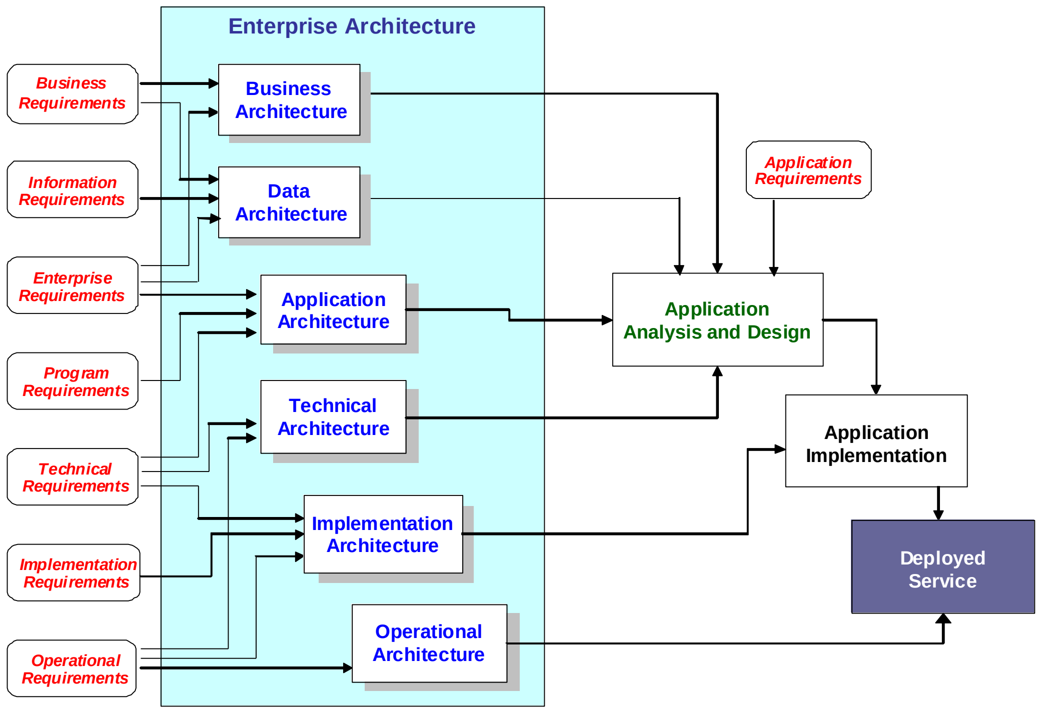
7.4 [redacted] in the Technology Architecture

[redacted]

8 Using Solution Architecture

Exhibit 8-1 illustrates the ‘Architecture Driven Design’ process. Each of the different BDATS sub-architectures (shown in rectangular, shadowed boxes) is driven by a set of enterprise requirements (shown in rounded rectangular boxes). We also introduced two additional architectures concerned with the implementation and operation of IT systems.

Exhibit 8-1: Architecture Driven Design



Each application has its own set of unique requirements. However, the application does not exist by itself; it exists in the context of the overall DOI enterprise. So, we use the business, information, application and technical architectures as a starting point for the analysis and design of a DOI application.

The application will fit into the architectural styles that are supported by the application architecture and technical architecture. If, for example, the application has a requirement for document scanning, it will use the patterns and services defined by the application patterns portion of the application architecture. The business processes and information required and supported by the application will conform to the business and data architecture of the enterprise.

Within this larger context, the application analysis and design are performed following the technology independent application architecture. When it is time to implement it, a technology choice must be made (unless the decision is made to use existing assets). The implementation architecture describes how to implement the application architecture on a specific technology choice (described by the technology architecture), taking into account tools and development processes. The platform-specific Reference Architectures illustrate the basic organization of an application's tiers and layers for J2EE and .NET applications.

After implementation, the application or service will be deployed into production. The Operational Architecture describes operational aspects such as monitoring, logging, backup, and replication and deployment aspects such as configuration and topology.

The following procedures are used (as appropriate) to apply the TSRA to a particular solution. Note that the following list is not meant to imply an order in which the steps must be executed – many of these steps may be performed concurrently. Also see Appendix B – Solution Architecture Artifacts for a list of the artifacts mentioned below.

- Business Architecture elements (see Chapter 3, Business Architecture):
  - Write a business description
  - Draw a Solution Overview Diagram
  - Create a Use Case Model
  - Define the functional requirements
  - Map the business, integration, and composite/custom patterns to the Solution Overview Diagram
- Data Architecture elements (see Chapter 4, Data Architecture):
  - Define the subject areas and information classes required by the solution
  - Map solution data to the DOI Conceptual and Logical Entity Relationship Model Diagrams, identifying common entities or classes
  - Create an Entity Relationship Diagram (ERD) for the solution
  - Create an Object Role Model (ORM) diagram for the solution
  - Determine whether an existing database, a COTS database, or a custom database is appropriate to use
- Application Architecture elements (see Chapter 5, Application Architecture):
  - Determine the appropriate application patterns based on the business, integration and composite/custom patterns
  - Define the Non-functional Requirements (NFRs)
  - Determine the application tiers based on the application patterns
  - Map the Service Component Reference Model (SRM) service components to the business functionality
  - Determine what business application components can be reused or purchased as COTS products
- Technology Architecture elements (see Chapter 6, Technology Architecture)
  - Decide the platform on which the solution will run
  - Determine the components of the technology-specific Application Reference Models required for the solution
  - Decide what infrastructure components can be reused or purchased as COTS products
- [redacted] elements (see Chapter 7, Security Architecture)
  - Determine appropriate user security levels and procedures
  - Define the appropriate data security levels

- Determine the [redacted] along with authorization and identification schemes
- [redacted]
- [redacted]
- [redacted]

At many points in this process, architects should ask whether components of the solution can be reused or customized from existing DOI applications or can be purchased as standard Commercial Off-the-shelf (COTS) products. There is no single moment when it becomes appropriate to ask this question. As soon as the solution overview exists, it may be obvious that an existing system could be used (as-is or customized) to satisfy part or all of a solution's requirements. It may not be until the solution is mapped in detail to a specific technology reference model, however, that it becomes clear that a standard COTS product could be used for one or more components. Whenever it is determined that an existing application or COTS product can be used in a solution, some of the above steps may be become unnecessary. For example, buying a COTS product that comes with a standard database structure obviates the need to create a solution ERD from scratch.

9 Appendix A – Glossary

A general list of terms and definitions follows this table of abbreviations and acronyms.

Abbreviation	**Definition**	**Comment**
AA	Application Architecture	
ACID	Add, Change, Inquire, Delete	Also known as CRUD (create, read, update, delete)
B2B	Business to Business Interaction	
[redacted]	[redacted]	[redacted]
[redacted]	[redacted] Architecture Repository	
BPM	Business Process Management	Allows composing finer-grained business and utility services into larger-grained business processes and provides transactional coordinating of multiple business services into atomic units of work (UOW)
BRM	Business Reference Model	
COTS	Commercial-Off-the-Shelf	
CTO	Chief Technology Officer	
CTOC	Chief Technology Officer Council	
DB	Database	
DEAR	DOI Enterprise Architecture Repository	
DOI	United States Department of Interior	
DRM	Data Reference Model	
EA	Enterprise Architecture	
EAI	Enterprise Application Integration	
ERD	Entity Relationship Diagram	
ESB	Enterprise Service Bus	
FEA	Federal Enterprise Architecture	
GUI	Graphical User Interface	
HTML	Hypertext Markup Language	
HTTP	Hypertext Transfer Protocol	Web-based protocol based on IP
IEA	(DOI) Interior Enterprise Architecture	
[redacted]	[redacted]	
IP	Internet Protocol	
TSRA	(DOI) Interior Solution Architecture	DOI Target Solution and Application Architecture
IT	Information Technology	
J2EE	Java 2 Platform, Enterprise Edition	Java version for developing and deploying enterprise applications
JMS	Java Message Service	
JSP	Java Server Pages	
LDAP	Lightweight Directory Access Protocol	
LOB	Line of Business	
MBT	Methodology for Business Transformation	IEA initiative
MSDN	Microsoft Developers Network	Reference site
MSMQ	Microsoft Message Queue	
MVC	Model-View-Controller	
.NET	Microsoft business strategy aimed at a convergence of personal computing with	

	the Web	
NFR	Non-functional requirement	
OCR	Optical Character Recognition	
ODBC	Open Database Connectivity	
ODS	Operational Data Stores	
OMG	Object Management Group	
ORM	Object Role Model (diagram)	
PRM	Performance Reference Model	
SDLC	Software Development Lifecycle	
SDLM	Software Development Lifecycle Management	
SLA	Service Level Agreement	
SLR	Service Level Requirements	
SOD	Solution Overview Diagram	
SRM	Service Component Reference Model	
TRM	Technical [or Technology] Reference Model	Technical Reference Model (FEA), Technology Reference Model (IEA)
UML	Unified Modeling Language	Specification from Object Management Group (OMG)
VB	Visual Basic	
XML	Extensible Markup Language	

9.1 Terms and Definitions

Application Pattern – partitioning of application logic and data together with styles of interaction between the logic tiers (i.e., the shape of an application) needed to satisfy a particular business pattern

Architectural Style – set of principles, elements, patterns, and constraints designed to meet a specific set of requirements within a specific scope

Architecture – a style and method of design and construction; the overall structure of a computer system, including the business processes and the required hardware and software

Business Area – BRM high-level category used to group LOBs: Services for Citizens (the purpose of government), Mode of Delivery, Support Delivery of Services, and Management of Government Resources

Business Pattern – generic description of rules identifying the interaction between users, businesses, and data

Business Reference Model (BRM) – FEA business-driven framework for describing the operations of the Federal Government (independent of the agencies that perform them) within business areas, lines of business, and sub-functions

Component – a constituent part of a system that can be separated from or attached to the system; a package of services and/or functions designed to work with other components and applications

Data Reference Model (DRM) – standardized framework by which data supporting government program and business line operations is described (Data Description), categorized (Data Context), and shared (Data Sharing); the DOI DRM ERD

Enterprise Architecture – comprehensive framework used to manage and align an organization's business processes, Information Technology (IT) software and hardware, local and wide area networks, people, operations and projects with the organization's overall strategy

Entity Relationship Diagram – graphical representation of entities, attributes used to describe them, and the relationships between them; principle data modeling tool for relational data model schemas

Fat Client – client-server architecture term for a client that local performs most of the application functionality, using a separate server only for data storage (if at all); politically incorrect term for thick client

Information Class – IEA DRM second-level data category within subject area (also referred to as a Super Type within the FEA)

Integration Pattern – pattern that does not automate a specific business problem but is used to integrate two or more business patterns

Line of Business (LOB) – BRM second-level category, a collection of sub-functions within a business area

Multi-channel – application architecture style that allows multiple access channels to one or more business services; see also Single Channel

Object Role Model diagram – graphic representation of object (entity types), relationships (fact types) between them, role the objects play in the relationships, constraints within the problem domain, and optionally examples (fact type tables)

Pattern - form, template, or model (or, more abstractly, a set of rules) which can be used to make or to generate things or parts of a thing

Performance Reference Model (PRM) – standardized FEA framework to measure the performance of major IT investments and their contribution to program performance

Rich Client – hybrid of thick and thin clients with more balanced client and server utilization with features and functionality of traditional desktop applications, which manage their deployment and updates and also intelligently connect to distributed services and data sources (Rich Client is the J2EE term for this type of client, e.g., implemented with Ajax – see also Smart Client)

Run-time Pattern – description of a set of nodes (i.e., groups of functional requirements) interconnected to solve a business problem

Service – work or duties done for others; in the context of SOA, any feature or business application function that may be used an IT component, including all necessary and relevant network and back-end resources that the feature or application function may use

Service Area – FEA TRM high-level category representing one of four technical tiers (Service Access and Delivery, Service Platform and Infrastructure, Component Framework, and Service Interface and Integration), a group of service categories

Service Category – FEA TRM second-level category of technologies and standards; a group of service standards by business or technology function served

Service Component – SRM low-level category defined as a self-contained business process or service with predetermined functionality that may be exposed through a business or technology interface

Service Component Reference Model (SRM) – FEA business- and performance-driven, functional framework that classifies Service Components with respect to how they support business and/or performance objectives within service domains, types, and components

Service Domain – SRM high-level category of services and capabilities that support enterprise and organizational processes and applications, differentiated by business-oriented capability

Service-Oriented Architecture – IT architectural concept that defines the use of services to support the requirements of IT users, in which nodes on a network make resources available to other participants in the network as independent services accessed in a standard way

Service Standard – FEA TRM low-level category of services and technologies supporting a service category, which may provide illustrative specifications or technologies as examples

Service Type – SRM second-level category within Service Domain that defines the business context of capabilities

Single Channel – application architecture style that only one access channel to one or more business services; see also Multi-Channel

Smart Client – hybrid of thick and thin clients with more balanced client and server utilization with features and functionality of traditional desktop applications, which manage their deployment and updates and also intelligently connect to distributed services and data sources (Smart Client is the .NET term for this type of client, implemented with Windows .NET Forms – see also Rich Client)

Solution Architecture – patterns and practices used to create a product or service that will allow a particular business task to be accomplished

Solution Architecture Pattern – a description of describes a particular recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution, where the solution scheme is specified by describing its constituent components, their responsibilities and relationships, and the ways in which they collaborate

Subject Area – BRM (Data Description) high-level data category, equivalent to a business activity within an LOB

Technical Reference Model (TRM) – FEA component-driven, technical framework categorizing (into Service Areas, Service Categories, and Service Standards) technologies to support and enable the delivery of Service Components and capabilities

Technology Category – IEA TRM high-level category, equivalent to FEA TRM Service Category; a group of technology standards by business or technology function served

Technology Reference Model (TRM) – IEA version of the FEA TRM, organizing Technology Specifications by Technology Category (FEA TRM Service Category) and Technology Standard (FEA TRM Service Standard)

Technology Specification– IEA TRM item, such as an application or product that falls within a Technology Standard

Technology Standard – IEA TRM low-level category, equivalent to FEA TRM Service Standard; a group of technology specifications and products

Thick Client – client-server architecture term for a client that performs most of the application functionality locally, using a separate server only for data storage (if at all); also known as a fat client

Thin Client - simple client program or device which relies on all data and most of the functionality of the system being located in the server

Use Case - specific way of using a system by performing some part of the functionality, constituting a complete course of action initiated by an actor and specifying the interaction that takes place between an actor and the system

Use Case Model – textual or graphic representation of all use cases for a solution

10 Appendix B – Solution Architecture Artifacts

The following table lists the artifacts for a solution complying with the TSRA.

Exhibit 10-1: DOI Solution Architecture Artifacts

Artifact	**Related Sub-architecture**	**Comments**	**Required**
Business Description	Business	Brief narrative	Required
Solution Overview Diagram (SOD)	Business	Enhanced in subsequent steps with business and integration pattern and service components overlays	Required
Use Case Model	Business	Diagram or text high-level with text documenting each use case and actor	Required
Functional Requirements	Business	List	Required
Solution Subject Areas and Information Classes	Data	List	Required
Solution data mapped to DOI entities or classes	Data	List	Required
Entity Relationship Diagram (ERD)	Data	Diagram	As needed for new development
Object Role Model (ORM) diagram	Data	Diagram	As needed for new development
Application Patterns and Tiers	Application	Diagram	As needed
Non-functional Requirements	Application	List	Required
SRM mapped to business functionality	Application	Diagram	As needed
Application Reference Model	Application/ Technology	Diagram	Required for COTS products and new development
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]
[redacted]	[redacted]	[redacted]	[redacted]

11 Appendix C – Service Oriented Architecture Roadmap

Service-oriented Architecture (SOA) initiatives are beginning to show up in many enterprises including the DOI. Not surprisingly, the hype around SOA causes many people within the enterprise to be skeptical of yet another 'silver bullet' technology. This skepticism is well founded in light of high profile failures of other 'wonder' technologies (BPR, objects, components…) to live up to the promises of reuse, improved quality and productivity.

On closer inspection, it turns out that many (if not most) of the reasons for those shortcomings was the failure of the IT organization to prepare and transform itself for the new technologies. And while new technologies have improved, and make it potentially easier to achieve the promised benefits, the root problems continue to be those of organizational transition. Against this backdrop, the promoters of SOA within DOI face battle-hardened skeptics of yet another new technology. These skeptics will grudgingly go along with giving SOA one chance to prove its value, but won't believe it until they see it in bits and bytes.

11.1 Rollout Strategy

The formula for a successful rollout at DOI is fairly straightforward: start small, empower the users, demonstrate value, incorporate lessons learned, roll out to a larger audience. This approach is designed to quickly demonstrate value and address the skeptics before attempting to set a major change in motion, and to allow the SOA team to learn and perfect what works best for their particular organization. It is critical to figure this out before attempting to expand the program, so that the rest of the organization can be successful with the new technology and processes. This approach also allows the SOA team to design the necessary supporting processes, and set the slow, organizational changes in motion so that they are ready when the other teams need them.

11.1.1 Pilot Project Phase

The first step is to choose an appropriate pilot project, one with the following characteristics: Important, but not on the critical path; achievable scope (4-6 months); demonstrable value; addresses a pain point. The best candidate projects will have a user for the new service lined up, provide a basic business application function that can be used across multiple applications, and that is discrete enough to run efficiently without adding complexity to itself or the applications that use it. Before getting too deep into the pilot project, it is important to have a vision of the goals of SOA for the DOI, and to have high-level architecture, business and information models in progress that lead the pilot project toward that vision.

Creating these models should take weeks, not months and each should serve a specific purpose. The architecture should lay out the high level technical direction and standards, and identify critical areas that need to be addressed by the pilot and the architecture should use the TSRA architecture as its foundation. The business model should identify high-level business processes and the underlying services that compose them, including the service being developed for the pilot. The business model should help to prioritize between candidate projects and identify reuse opportunities and strategies. The information model should identify the semantics at the service and enterprise levels that are required for the identified service to support the enterprise business process, and be usable across the set of other identified potential uses. The business and information models should build on the eGov service models being developed as part of the eGov project.

The pilot project has three important goals. First, to give the core SOA team the opportunity to learn the 'what and how' of implementing a service at the DOI (Each organization has slightly different requirements, environment and culture which will dictate the best approach for them). Second, the pilot project needs to deliver a working service. This goal requires a well-chosen pilot, and an achievable scope and timeframe. Four to six months is typically a good timeframe for the design, implementation, and testing of the initial service. Depending on the goals of the project, the service may or may not be deployed into a production environment. Four to six people is also a good size for the core team. This will include the architect, analysts, designers, implementers, testers, and project leader/manager. Although this count does not include the subject matter experts from the business, it is important to get them involved in the design of the service, as well as the business and information models. Last but not least, the pilot project is intended to demonstrate the value of SOA. This means that you must keep track of the time and effort spent on implementation, have reasonable projections for how SOA delivers value, and have realistic business scenarios to show how the service will be reused and add value outside of the initial application.

11.1.2 First Adopter Phase

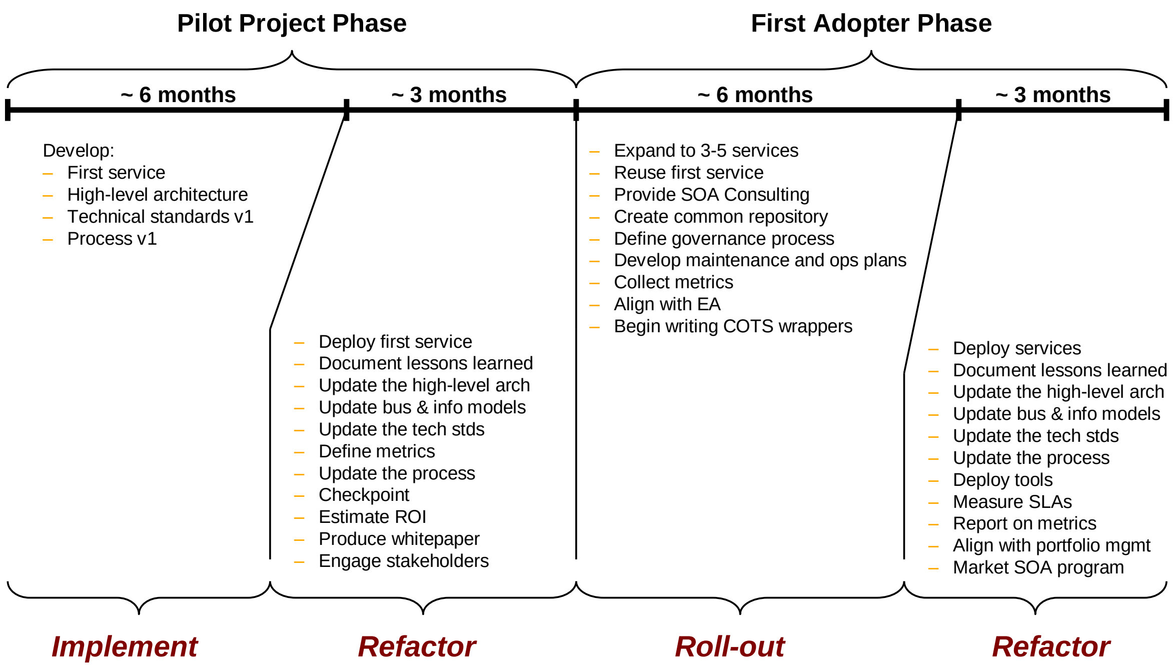
The goal of the next phase in the SOA rollout will be to expand SOA use and development to additional projects. We call this the 'First Adopter" phase. Exhibit 11-1 shows a sample SOA Rollout Plan for the pilot project and the First Adopter phases.

After the pilot project is complete, and before the First Adopter phase starts in earnest, it is advTSRAble to take two or three months to regroup, plan for the next phase, identify metrics, incorporate lessons learned, market your SOA efforts, and identify the next set of services and organizations. Expanding the use of services is a combination of marketing the value of SOA and identifying good candidates for the First Adopter round of service implementations. As with the pilot, these services should be valuable and achievable, but also have some additional requirements.

- The new services should be related to the pilot service implementation so that at least one of the new services can be combined together with the initial (pilot) service in a business application. This will begin to demonstrate the potential of SOA in creating higher level, higher value processes.
- One additional business user should be identified for the initial service. This will help to demonstrate the value of services as reusable assets. To the business users and sponsors, consistent processing of a common business application function across multiple applications may be a more important outcome of reuse than reducing development time or costs, so look for opportunities to solve an existing incompatibility problem.

Before selecting the follow-on users (applications) and service implementations, the core team should update the business and information models to reflect that actual implementation of the initial service, and to incorporate any lessons learned. These models will be carried forward and expanded during the First Adopter phase, so they should be correct, and complete enough to help narrow down the selection of the next set of services. Again, these models should take weeks, not months of effort to update and develop (avoid analysis paralysis!). Having a related set of services allows the team to focus on developing these models in more detail, within a limited scope, and to postpone the issues of expanding into new organizational boundaries until you have built up momentum and support for the SOA initiative. One of the key challenges and goals of the information model is to define the service semantics, at a business level, so that they span and integrate several related services and processes. The First Adopter phase of the rollout should provide the core team an opportunity to approach this challenge within a controlled environment.

Exhibit 11-1: Sample SOA Rollout Timeline



In order to expand development, the core SOA team must make it possible for the new development groups to be successful with their attempts at SOA. The two most important components of this are the SOA architecture, and implementation support. The architecture defines exactly what services are for the particular organization, including the types of services, interface and interaction style, required interfaces (such as management), how to use the service technical infrastructure, how to publish and find services, etc. Each time the implementation of services is expanded to more developers, these aspects have to become more explicit, more clear, more foolproof. The pilot phase is where these details are first worked out. For the First Adopter phase, they have to be documented well enough to enable the new service developers to implement them correctly, but they don't have to be perfect, the implementation support will fill in the holes. (Note that after the First Adopter phase, you will again incorporate the lessons learned and update the SOA and process before starting the Rollout phase).

Implementation support is a critical activity for the core team during the First Adopter phase. For the SOA initiative to be successful, the First Adopter teams have to be successful implementing and using services. The core team must monitor and assist these implementation teams to help them work through issues and get their designs and implementations correct. This will also give the core team first hand exposure into how well the architecture and guidelines are working.

During the First Adopter phase, the core team will not be implementing any new services, although they will be updating the initial service to support the new business user. But, the core team will be very busy. In addition to the implementation assistance, they will have to work out the issues of service lifecycle and governance. In particular, they will have to plan for who is responsible for services. At a technical level, that means how will services be versioned? What will be the development and runtime repository support for services? At a governance level, it means who owns, defines monitors, and authorizes new services or changes to existing services? Who is responsible for the provisioning and maintenance of services? How is the alignment of services with the business and EA maintained? What will the governance structure and organization be? Note that although governance is responsible for insuring conformance, the thrust of the governance organization should be to help teams build conforming applications from the start, not to check up on them at the end. A plan for all of this needs to be developed during the First Adopter phase, so that it can be executed while the next phase, Rollout, begins, and be in place by the time it is needed.

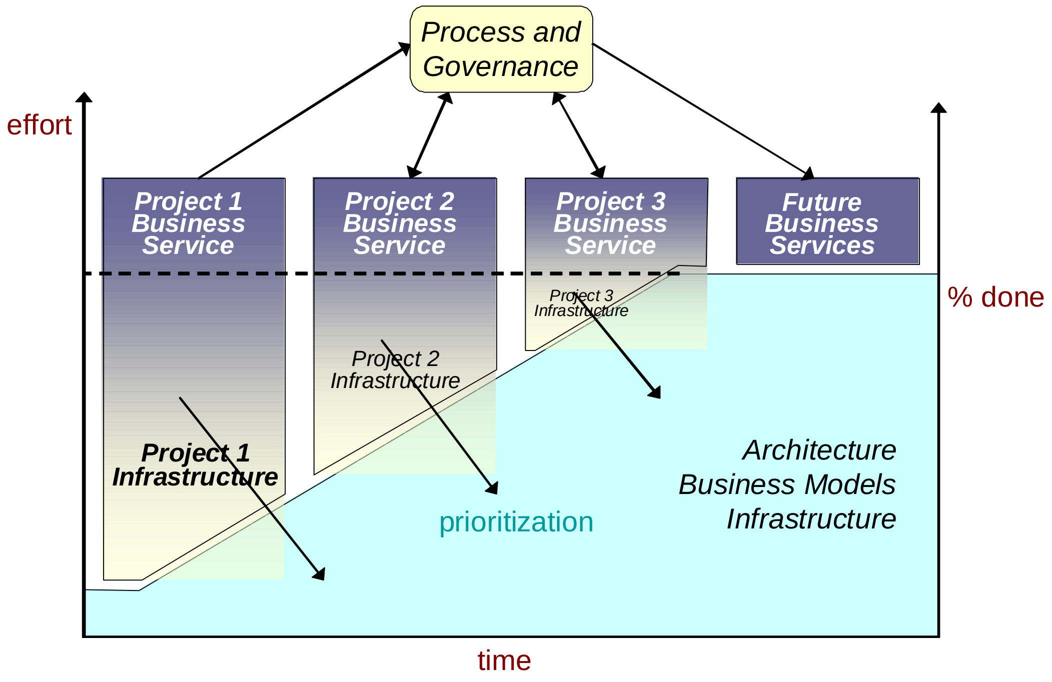
The most important activity during the SOA rollout process is the metrics. That means defining what to measure, measuring it, and reporting on it. During the pilot phase, you will keep informal metrics, but be thinking about what data will make the best case for SOA value. During the three-month regroup, the core team will formalize those metrics and include them in the SOA and development guidelines. During the First Adopter phase, the development teams will collect these metrics (with any necessary assistance from the core team). At the end of the First Adopter phase, the core team will collate and report on the metrics. This will be the single most important evidence of the value of SOA that you can present to management, the sponsors, and the skeptics. Nothing is more convincing than real, hard numbers! The importance and value of these metrics cannot be overstated.

SOA can provide significant advantages in moving DOI toward achieving its goals, but provides significant challenges in adoption. A ‘big bang’ approach will not be successful. Instead, SOA has to be eased into the organization while DOI learns the best ways to implement and incorporate it. The task of the core team is not to figure out what SOA is, but to figure out how to make the rest of the DOI development organization successful implementing SOA. At the same time, SOA has to deliver and demonstrate value to the business, by implementing well-chosen services, and collecting convincing metrics.

11.1.3 Phased Implementation Approach

Like other architectural efforts, SOA needs to be rolled out while DOI is continuing to deliver projects and value. Exhibit 11-2 illustrates how each project contributes both to the business and building up the necessary architecture and infrastructure to sustain SOA at a steady-state.

Exhibit 11-2: Phased SOA Capability Attainment



The trapezoidal box at the bottom represents the architecture, infrastructure and business models that make up DOI SOA capabilities and assets. At the beginning of the first project, there is very little of this is place. So, the first project has to build some of it themselves, take short cuts in other areas, and do without certain aspects in order to deliver a service that provides business value in a reasonable timeframe. At the end of Project 1, some aspects of the infrastructure built for Project 1 will be rolled into the overall infrastructure (as described in the refactoring activities earlier).

The next project (Project 2) that comes along will have more of the architecture, models and infrastructure in place, but still not all of it. So, like the first project it will have to build some and do without other aspects of it. Again, at the end of Project 2, parts of its infrastructure can be used to contribute to the overall architecture. At the beginning of each project, a prioritization is done to choose what will likely be rolled into the architecture based on both the needs of the particular project, and the larger enterprise architecture context (for example, a capability that can be used by every subsequent project would be a good candidate for building sooner, rather than later).

This cycle is repeated again and again until eventually, the steady state of the architecture is achieved. This typically takes between 3-6 project cycles, depending on the length/size of a project. At that point, each new project only needs to build its business services because all the rest of the infrastructure and architecture is in place. Some adjustments and enhancements to the infrastructure will always be needed, but they will be a relatively minor activity. Thus, when the steady state is reached, the amount of time and effort required for new projects is dramatically reduced.

The oblong box at the top of Exhibit 11-2 represents the development process and governance capabilities to support building and maintaining services. A similar process is followed to build up these capabilities. The first project will start with very little process or governance. (Of course, very little governance is needed before there are any services). Some of the lessons learned and processed developed in the first project will be carried forward and used by the second project. At the same time, governance capabilities will start to be developed (as described earlier in the First Adopter Phase). Subsequent projects will both use and contribute to the overall process and governance capabilities until the steady state is achieve at which time new services will be built conforming to the defined standards, processes and governance.

11.1.4 Rollout Summary

The following list summarizes the major activities of an SOA Rollout for DOI:

- Develop an SOA Roadmap for the next 2+ years similar to the sample above
- Establish the high level SOA for DOI
    - Align with business goals, Enterprise Architecture and eGov
    - Determine important criteria
    - Specify what a service is and how to implement on J2EE and .NET
- Start with well chosen initial projects (submitting Form 300's) to build expertise and acceptance
    - Important but not on the critical path
    - Achievable scope
    - Addresses a pain-point
    - Demonstrable ROI
- Incorporate lessons learned into architecture and approach. Continue architecture and infrastructure implementations
- Initiate Service Governance activities and organization
- Identify new applications suitable for service implementation. Expand team and expertise
- Re-evaluate business needs to decide on next projects. Continue to 'refactor' service, architecture and infrastructure
- Establish service repository
- Roll out to an expanded group of developers
- Create service implementation frameworks

The details of the technologies, platform and enterprise capabilities that are implemented in this phased approach are describe in the following sections.

11.1.5 SOA Maturity Stages

Every organization that adopts SOA goes through a maturation process while they gradually build up capabilities. The model below provides some common maturity phases that most organizations will experience. Today, DOI is somewhere between the Skunkworks and Independent Production Services level of maturity.

- **Skunk works –** Some developers have independently decided to experiment with web services as a way to achieve some sort of integration. This phase is characterized by:
  - One off projects
  - Experimentation with web services
  - Web services used for integration of heterogeneous systems
  - No service contracts or other SOA concepts
- **Independent Production Services –** Web services have been built for one or more projects and have been moved into production. Now, the organization needs to understand how to manage and support them. It typically takes 6-12 months for an enterprise to advance from this to the next level. This maturity level is typically characterized by:
  - Move to production of limited web services
  - Simple integration with partners
  - Wrap legacy systems with web services
  - No coordination between projects or services
  - No mechanisms in place for support or maintenance
- **SOA Program Initiation –** There is an official recognition of the importance of SOA and the need to manage them. This leads to the introduction of an SOA program and the recognition that an SOA architecture needs to be developed. Some parties have an understanding of the overall enterprise context that is required to achieve the long terms goals of the SOA program. It typically takes 12-24 months for an enterprise to advance from this to the next level. This maturity level is typically characterized by:
  - Limited/controlled SOA based project development
  - Begin architecture program
  - Begin business model
  - Begin information model
  - Establish governance
  - Establish reuse program
  - Establish service management capabilities
- **Enterprise Rollout –** The SOA rollout is well under way. The architecture and infrastructure has reached a level where it can support mainstream development activities. Service reuse is fully supported by programs, process, repository, versioning and taxonomy. It typically takes 12-24 months for an enterprise to advance from this to the next level. This maturity level is typically characterized by:
  - Most new projects will be SOA based

- Expanded development across enterprise
- Governance in place
- Management in place
- Well established taxonomy and reuse program
- Versioning capabilities
- Service repository supports both development and runtime

- **SOE –** The Service-Oriented Enterprise. Business processes are implemented by combining services together based on Business Process Models. Business Process management systems are well in places. Processes and service design is driven by the business and information architectures. The SOA architecture, infrastructure and frameworks are fully mature and in place. A large library of services is in place so that most projects can be built based on the existing services. This maturity level has the following characteristics:
  - Almost all projects will be SOA based
  - BPM systems are mainstream
  - New processes are constructed by ad-hoc service integration (i.e. no apriori plan existed to use the services together)
  - Service repository of 100+ services
  - Agility to create new processes and services quickly
  - Time to market for new applications is greatly reduced
  - Overall IT maintenance budget is reduce
  - Improved Quality of services

DOI should begin an architecturally based pilot project rollout and the initiation of an SOA program as soon as possible.

11.2 Technology and Standards

The underlying technology for SOA should be based on Web services. Although they are not necessarily required, they make the implementation of SOA much easier and inherently support the concepts and characteristics of services.

[redacted]

[redacted]

11.2.2 Web Service Standards

One of the requirements for services at DOI is to support development on multiple platforms. Another requirement is for those services to be able to interoperate. Because of the variation in web service specifications and versions and the number of ways that they can be used, DOI needs to adopt a set of standards and practices. Luckily though, DOI does not need to figure it all out themselves. The Web Service Interoperability (WSI) organization provides a set of profiles and scenarios that meet the requirements of the DOI. These include:

- **Basic Profile V1.1** – The WS-I Basic Profile provides interoperability guidance for a core set of non-proprietary Web services specifications along with interoperability-promoting clarifications and amendments to those specifications, including:
  - SOAP 1.1
  - WSDL 1.1
  - UDDI 2.0
  - XML 1.0 (Second Edition)
  - XML Schema Part 1: Structures, Part 2: Data types
  - HTTP 1.1
  - HTTP over TLS Transport Layer Security, HTTP State Management Mechanism
  - The Transport Layer Security Protocol Version 1.0
  - Internet X.509 Public Key Infrastructure Certificate and CRL Profile
  - SSL Protocol Version 3.0
- **Simple SOAP Binding Profile 1.0** – The Simple SOAP Binding Profile describes the serialization of the envelope and its representation in the message
- **Attachments Profile 1.0** – Complements the Basic Profile 1.1 to add support for interoperable SOAP Messages with attachments
- **Sample Architecture Usage Scenarios 1.0** – Usage Scenarios define the use of Web services in structured interactions, identifying basic interoperability requirements for such interactions and mapping the flow of a scenario to the requirements of the Basic Profile.
- [redacted]
- [redacted]
- [redacted]

Other Web service standards are evolving to support Transactions, Reliable Messaging, Routing, etc. The use of these standards should be isolated within the underlying foundations services. For example, the transaction service should determine the best use of WS-Coordination and WS-Transaction standards to meet the needs of DOI. The Workflow service should determine the requirements and use of WS-ReliableMessaging and WS-Routing. Isolating the use of these evolving standards to a foundation service also isolates any changes that will be required as the standards evolve. Only the foundation services will need to be updated, not every application that uses them.

11.2.3 Other Standards

A primary goal of SOA is to create reusable services, but how do you describe a reusable service? What aspects need to be described to support development and runtime? The **Reusable Asset Specification** (**RAS**) that is now part of UML 2.0 provides a complete (perhaps too complete) definition of what a reusable asset is and how to describe it. DOI should use this as a starting point for its definition of a service as part of the SOA.

11.3 The SOA Platform

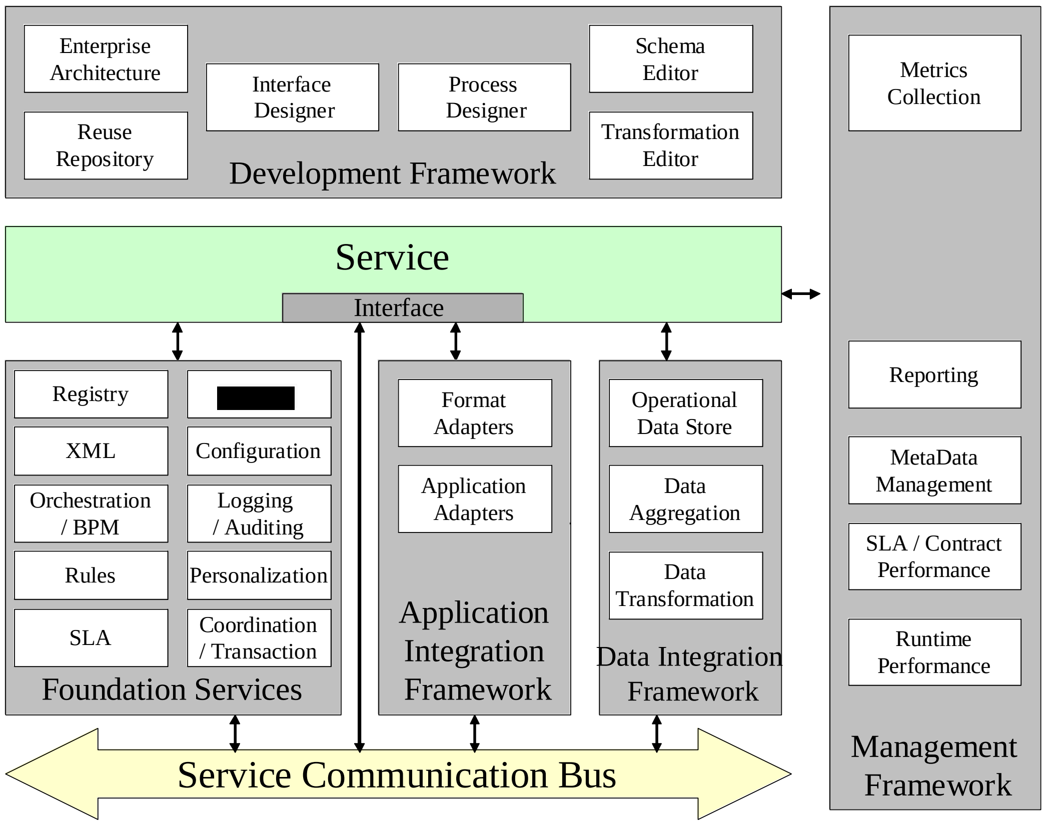
Web services are an important basis for SOA, but much more is required to achieve the higher levels of maturity outline previously. Exhibit 11-3 illustrates the capabilities required of a complete SOA platform, which supports the following areas:

- Basic Communications Infrastructure (Service Bus)
- Technology and Infrastructure Services
- Application Integration
- Data Integration
- Management
- Development

At the bottom of the diagram is the Communications Bus. This provides the basic connectivity for the different services. In the middle of the diagram is the service itself that the SOA is concerned with creating and supporting. The service is connected to the service bus through its service interface (Tools generate the code required for the connection based on the interface.). An important special case is services that are created by wrapping existing applications (application integration). These services go through the same interface mechanism. Business services use a variety of infrastructure services to assist in their runtime implementation. Similarly, the development framework provides a set of capabilities to assist in design, development and debugging of services. The data integration framework provides business services with access to enterprise data resources. Finally, the management framework supports the operations of the infrastructure and services.

Note that the platform example is meant to be representative of most SOA applications at DOI, but does not present an exhaustive list of all possible services.

Exhibit 11-3: SOA Platform



11.3.1 Communications Service Bus

The communications service bus is essentially provided by web services, but can use other interface protocols. The primary responsibilities of the service bus are to provide:

- Basic communications infrastructure to connect service consumers with service providers.
- Dynamic Discovery to allow service providers to register their ability to provide specific services and to allow service consumers to find them.

The best practice for all communications is to use pure XML. Using the XML standard increases the potential for asset reuse and ensures compatibility with most, if not all, types of services.

11.3.2 Foundation Services

The foundation services provide common capabilities that are used in the construction of business services, independent of any business domain. Overall, these services include:

- **XML capabilities -** these include: Message Formatting, Message Parsing and Data Transformation.
- **Registry** – Although the registry is often physically provided by the communications bus, it is usually packaged as an infrastructure service to provide location independence.
- **BPM / Orchestration** – the business process management service provides a mechanism for defining and executing business process models that create higher level business or enterprise services from lower level process and services.
- **Coordination and Transactions** – provide the capabilities to coordinate multiple services together into an atomic unit of work.
- **Rules -** allows business rules to be expressed by business analysts, independent of code, and then applied to different processing scenarios. Rules are externalized and can be updated easily and frequently. For example, business rules can be used to implement regulatory requirements that vary widely by state or country and are subject to constant revision.
- **Personalization** – Provides the storage and retrieval of information relating to service consumers for the purpose of 'personalizing' their interaction with service and the enterprise.
- **Configuration -** provides a common mechanism for defining, storing and retrieving application configuration information. Enables applications to share common configuration parameters.
- [redacted]
- **Logging and Auditing -** provides common mechanisms for logging information and errors and auditing actions. The services may also provide common utilities for reading and reporting on logs and audit trails. In addition to the benefits of reduced development and operations costs, common logging and auditing allows information from business services that span several systems to be correlated.
- **Service Level Agreements (SLA)** – provide capabilities that allow services to define, monitor and measure quality-of-service characteristics such as throughput, performance and availability.

11.3.3 Data Integration

The data integration framework provides a standard way (i.e., an Enterprise Data Strategy) to aggregate and transform information that is collected from a variety of existing enterprise data sources and used by business and enterprise services. This includes:

- **Data Transformation -** provides a metadata-driven mechanism for transforming data from one format to another. A GUI interface is usually provided to a 'transformation designer' tool that can be used to define the data mappings.
- **Data Aggregation** – provides a mechanism for combining data from multiple sources into a single, new entity.

- **Operational Data Store (ODS)** – provides run-time access to the aggregated and transformed data. It is used primarily by the enterprise business processes and business services. The ODS may be real, or virtual.
- **Data Warehouse** – database geared towards business intelligence requirements, integrating data from various operational systems and typically loaded/updated at regular intervals.
- **Data Mart** – small version or subset of a Data Warehouse.

11.3.4 Application Integration

Successful SOA implementations over the years have shown that providing a service-based interface to existing applications leads to more flexible and cost-effective integration than traditional point-to-point EAI solutions. DOI relies on EAI technology to provide the application integration capabilities. This requires the following components.

- **Application Adapters** – translate between the native communications mechanism of the application and the communication mechanism of the target system, in this case SOA.
- **Format Handlers** – provide standard mechanisms for handling the different data transfer formats. In many systems, fairly rudimentary methods are used to transfer data and initiate requests, including file transfer, ftp, batch, etc.

11.3.5 Management

All enterprise systems need to be managed and that includes SOA based applications. However, the scope of SOA is frequently much larger than the scope of single applications, and the issues of management need to be addressed in the architecture and the platform. In particular:

- **Run time management** – provides mechanisms for managing services including starting, stopping, failing over, collecting statistics, determining version information, etc.
- **Contract / SLA performance** – provides a mechanism for managing the contract and service level agreements associated with specific services and consumers. An SLA is an important concept for managing performance, but it is fairly useless if it cannot be measured, enforced and reported on.
- **Metadata management** – provides a mechanism for managing the metadata artifacts associated with many different technologies and infrastructure services. In addition, it allows the relationships between metadata to be managed and exploited.
- **Metrics collection** – provides a mechanism for defining metrics and collecting the data associated with them. DOI has specific business and IT goals; it needs the capabilities to actually measure performance against those goals.
- **Reporting** – provides flexible mechanisms for the generation of reports related to the different areas of management.

11.3.6 Development

Last, but not least, is the actual development of services and enterprise processes within the context of DOI and SOA. The following facilities are required:

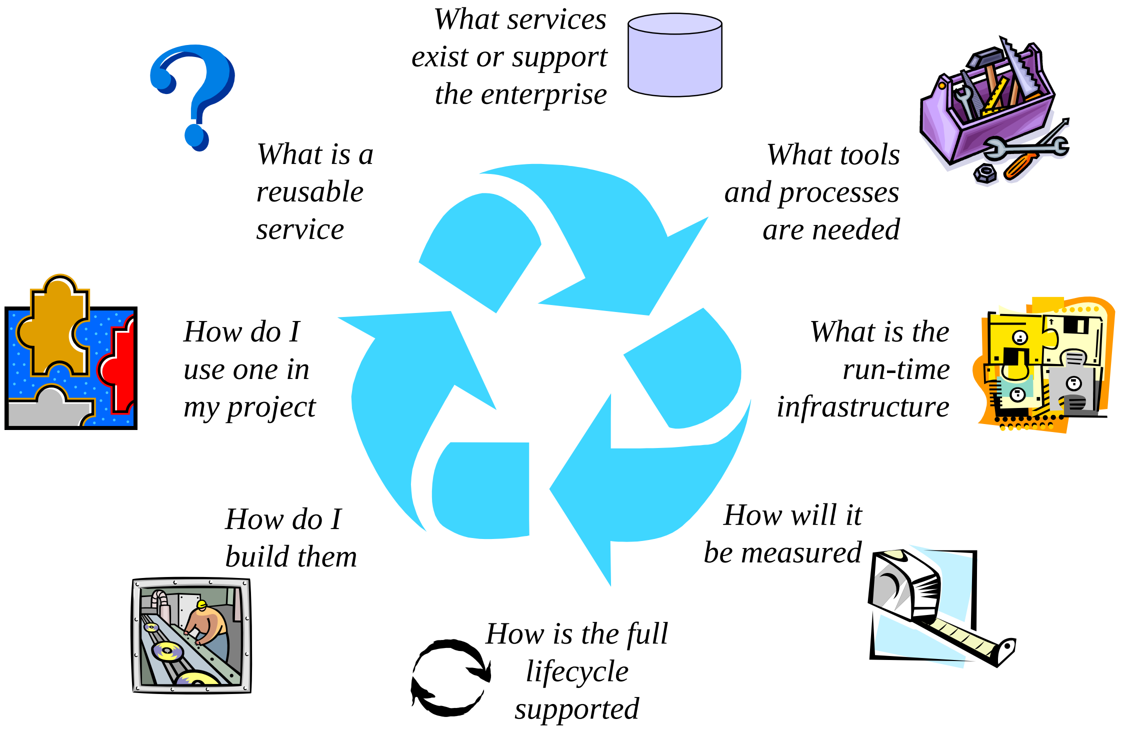
- **Enterprise Architecture** – defines the context, scope and relationships of applications, services, information, technology and infrastructure for DOI. This viewpoint is necessary in order to achieve the goals of independently developed services that can be combined to create higher-level business value. In particular, the business and information aspects of the enterprise architecture are required to direct the design of specific services and entities.
- **Schema Editor** – provides a visual tool for the creation of XML schema that is used to define the format of service messages.
- **Transformation Editor** – provides a visual tool to map the transformation of data from one format to another.
- **Interface Designer** – provides a higher-level paradigm for specifying service interfaces. The necessary infrastructure-level artifacts are then generated from the interface design.
- **Process Designer** – provides a visual tool for the definition of business processes, which are executed using the orchestration or BPM facility.
- **Reuse Repository** – provides a design time mechanism for publishing services and their related artifacts, searching and finding services, managing service lifecycles and managing and measuring service usage. A fundamental objective of SOA is the creation and reuse of services. However, reuse doesn't just happen, it needs to be managed and facilitated. Note that the runtime requirements of service publication and discover (as implemented by the registry) are significantly different than the design time requirements for service lifecycle management. A reuse repository should be incorporated into the development environment for DOI.

11.4 Enterprise Capabilities

The maturity model hinted at a requirement for enterprise capabilities to support an SOA initiative. This section list the key capabilities needed for a successful SOA program at DOI:

- **Architecture -** We have discussed SOA in depth already. However, since it is key to achieving an SOA, we list it as a required capability.
- **Reuse repository and program –** Reuse of services is absolutely essential to SOA at DOI. But we have learned through painful experience that reuse doesn't just happen by itself. It needs to be carefully managed. Exhibit 11-4 illustrates the following issues that have to be considered for an effective reuse initiative.

Exhibit 11-4: Major Issues of a Service Reuse Program


What services exist or support the enterprise
What is a reusable service
What tools and processes are needed
How do I use one in my project
What is the run-time infrastructure
How do I build them
How will it be measured
How is the full lifecycle supported

- **What services already exist** – A repository of services is essential. The repository needs to support the development time activities of searching for a services and determining if the service is appropriate. This is not the same as a runtime (UDDI) repository. There are several major repository products available that support the RAS.
- **What is, and is not reusable** – A lot of business functionality can be structured as services, but not everything should be implemented as a reusable service. The architecture needs to clearly define what is and isn't reusable and provide developers with guidance on making these decisions.
- **How to use reusable services** – It must be clear how services should be used in an application (e.g. what standard features they have, what required interactions there are, etc.)
- **How to build reusable services** – The architecture must define the structure of a reusable service and how to build one.
- **How to evolve and maintain services** – The reuse program must describe the complete lifecycle of services.
- **How will service reuse and quality be measured** – Metrics need to be defined, collected and reported on as part of the reuse initiative.
- **The application infrastructure required to support services** – A service is useless by itself. Rather, it is designed to fit into a specific environment. This environment (infrastructure) and the services provided by it must be described by the architecture.

  - **The development environment / frameworks / infrastructure / tools required to support service reuse** – It is not enough to describe what reusable artifacts are, the architecture must enable the easy and efficient creation of those artifacts.
  - **The metrics for measuring program success** – Reuse is only effective if it meets the business goals that drive the reuse program. The architecture must choose metrics to demonstrate those goals and a method for collecting the metrics.
  - **How will versioning work** – Although this is part of the lifecycle, it is difficult and important enough to call out separately. There will be multiple versions of services. Both policy and infrastructure need to be in place to support it.
  - **Service Taxonomy -** The taxonomy is based on the roles and responsibilities that different services exhibit, and maps directly to the business model. The taxonomy helps developers of services understand how their functions fit into the big picture and help minimize redundancy of responsibility. The taxonomy also helps consumers of services understand what set of overall services exist and the patterns of application construction that are easily supported. In addition, the taxonomy suggests a hierarchy of services. The service taxonomy should be based on the service model being developed for the eGov project.
- **Metrics –** Metrics must be in place to measure the SOA initiative. This includes:
  - Defining metrics to measure the goals and issues of SOA value, adoption, performance, etc.
  - Having a mechanism to incorporate metrics into service implementations, both at the runtime and development levels
  - Having a mechanism to collect and report on metrics
  - Using the metrics to drive a program of continuous improvement
- **Service Management –** As with any other enterprise systems, management and operations are extremely important. SOA will introduce new requirements into the management systems at DOI. These systems must be able to provide answers to the following questions:
  - Are your services up and running?
  - Are the right consumers accessing the right services?
  - What are the usage and performance characteristics?
  - Are you providing the required quality of service?
  - How do you fix things when something goes wrong?
  - How do you specify policies and characteristics?
  - How do you provide graceful upgrades?
- **Service Level Agreement monitoring and reporting** – Service level agreements provide an important way to manage and insure Quality of Service. The implementation of SLA at DOI should include:
  - Defining the technical performance characteristic of services, such as up-time, response time, and throughput
  - Defining business related performance metrics for services
  - Having a mechanism to incorporate SLA metrics into service implementations
  - Having a mechanism to collect and report on SLAs
  - Notification (runtime and reporting) when performance falls near or below required SLA levels.

  - Proactive runtime intervention to ensure SLAs
- **Service Oriented Integration Center Of Excellence (SOI CoE)** – The SOI CoE provides a centralized group that specializes in the intricacies of integration. The SOI CoE is responsible for:
  - Creating an enterprise wide integration infrastructure as part of the overall DOI technical infrastructure
  - Keeping the integration aspects of the TAA up to date
  - Implementation support and assistance
  - Lifecycle management of common integration services
  - Facilitating reuse of integration components

11.5 SOA / ESB Patterns[5]

IBM ESB usage patterns provide a means for describing and defining interactions and component topologies at the system or solution level and help us to see how and where the abstract concepts that we have been describing can be applied to specific implementation scenarios. Patterns enable and facilitate the implementation of successful solutions through the reuse of components and solution elements from proven successful experiences. IBM's patterns for e-business provide one such example and, with specific relevance to the ESB, introduce a set of collaboration patterns that design or describe broad organizational relationships among applications and a set of interaction patterns that describe required behavior in greater detail.

The fundamental concept in this case is that of the broker application pattern, in which distribution rules are separated from applications, enabling great flexibility in the distribution of requests and events and reducing the proliferation of point-to-point connections, thereby simplifying the management of the network and system. This basic pattern appears in several variations, and we will briefly consider each of these variations in this section.

[5] http://researchweb.watson.ibm.com/journal/sj/444/schmidt html

11.5.1 Service- and Event-Routing Pattern

A request or event is distributed to at most one of multiple target providers (see Exhibit 11-5). Examples may include simple service selection based on context or the content of the request or on more complex models, in which service requests can be routed to particular systems based on availability, workload, or detection of error situations. Service selection may involve the lookup of appropriate service providers in a service registry.

Exhibit 11-5: Service- and Event-Routing Pattern

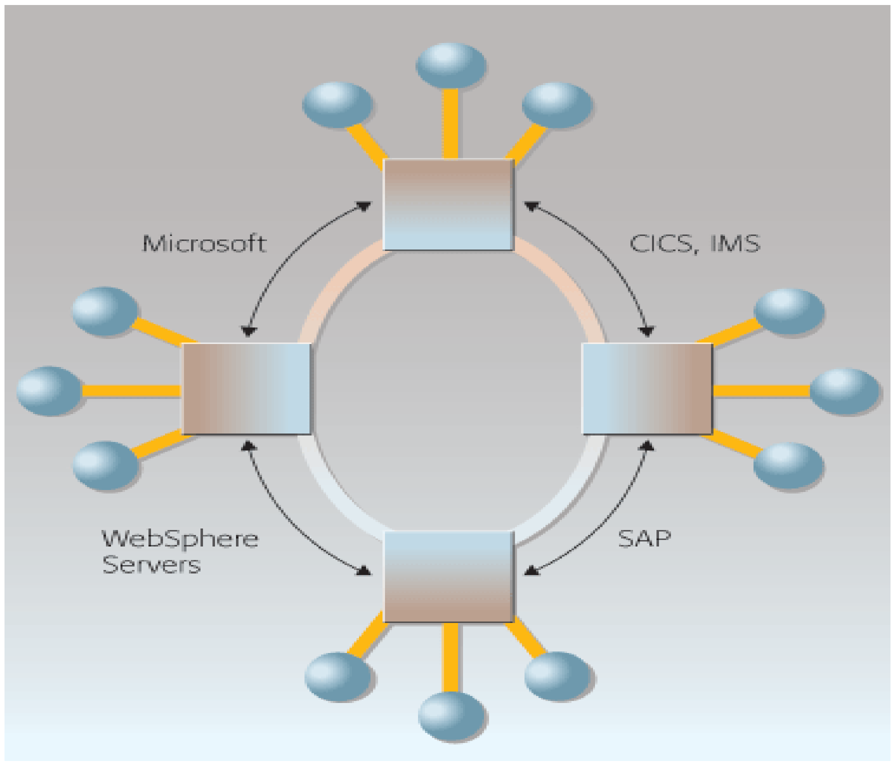


Figure 7
Service- and event-routing pattern

11.5.2 Protocol Switch Pattern

A routing pattern in which requestors and providers use differing network protocols (see Exhibit 11-6). Examples may include simple mapping of SOAP/HTTP requests onto a more reliable SOAP/JMS infrastructure or mapping between JMS and non-JMS applications.

Exhibit 11-6: Protocol Switch Pattern

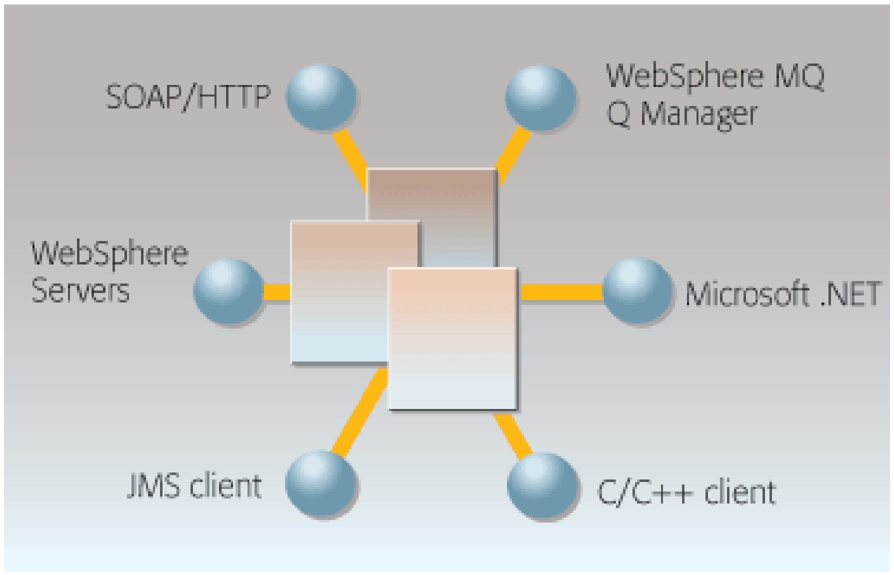


Figure 8
Protocol switch pattern

11.5.3 Proxy or Gateway Pattern

A variant of the routing pattern (or protocol switch) which maps service interfaces or endpoints, [redacted] he proxy may also support dTSRAggregation (and subsequent reaggregation) of a single request into multiple component subrequests. Examples of this pattern include service portals, in which a single point of contact is provided for multiple services and the details of "internal" services may be hidden from the service requestors.

Exhibit 11-7: Proxy or Gateway Pattern

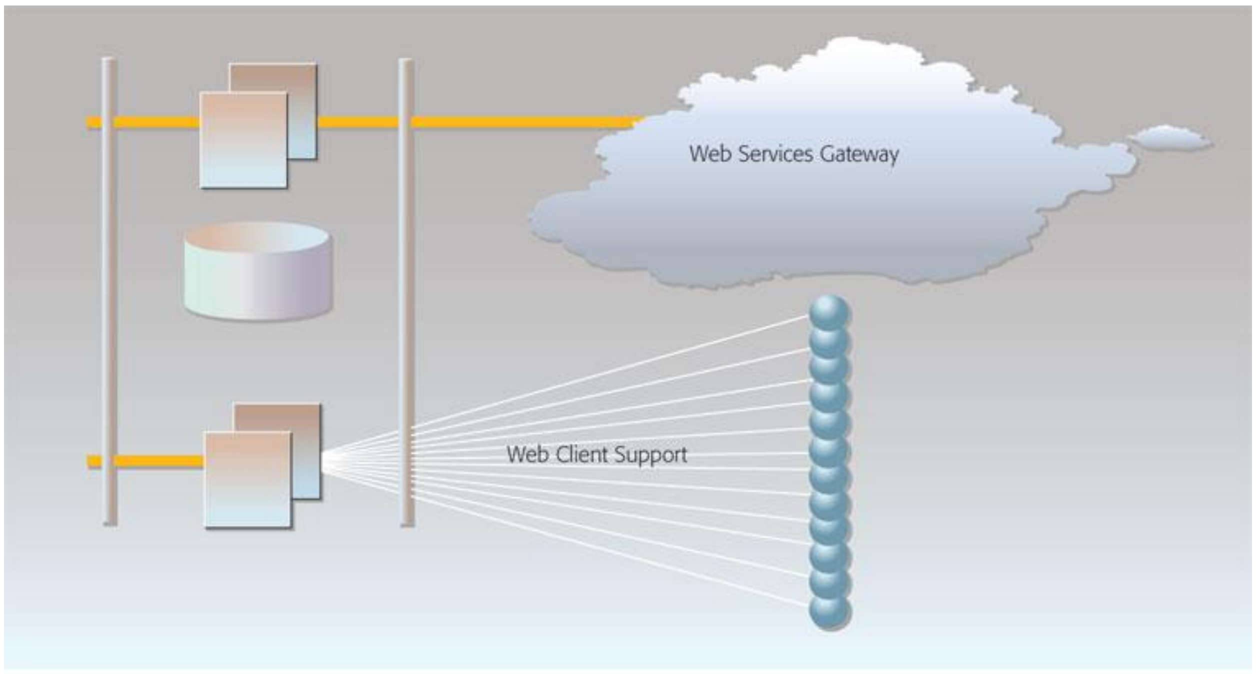


Figure 9
Proxy or gateway pattern

11.5.4 Event Distribution Pattern

Events may be distributed to more than one target provider, based on a list of interested parties that is managed by the ESB (see Exhibit 11-8). Services that wish to be notified of such events may be able to add themselves to the interested-parties list. An example of this pattern would be the distribution of business events based on CBE through the common event infrastructure.

Exhibit 11-8: Event Distribution Pattern

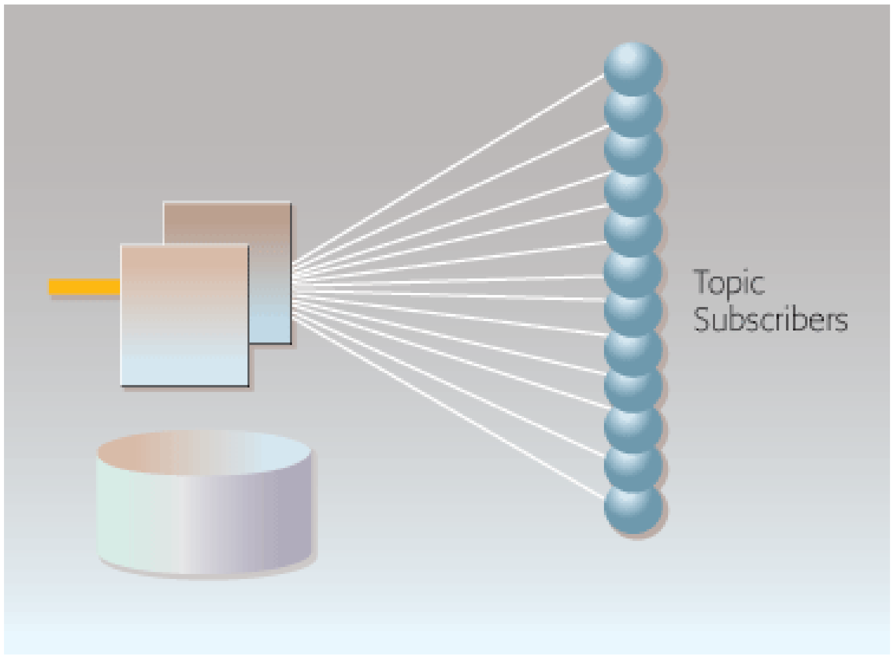


Figure 10
Event distribution pattern

11.5.5 Service Transformation Pattern

Requestors and providers use different service interfaces, and the ESB provides the necessary translation (see Exhibit 11-9). This pattern exposes new service interfaces without requiring change or modification to an existing application or service. It may also be used when multiple providers support the same business function but provide different interfaces, allowing this difference to be hidden from the service requestor.

Exhibit 11-9: Service Transformation Pattern

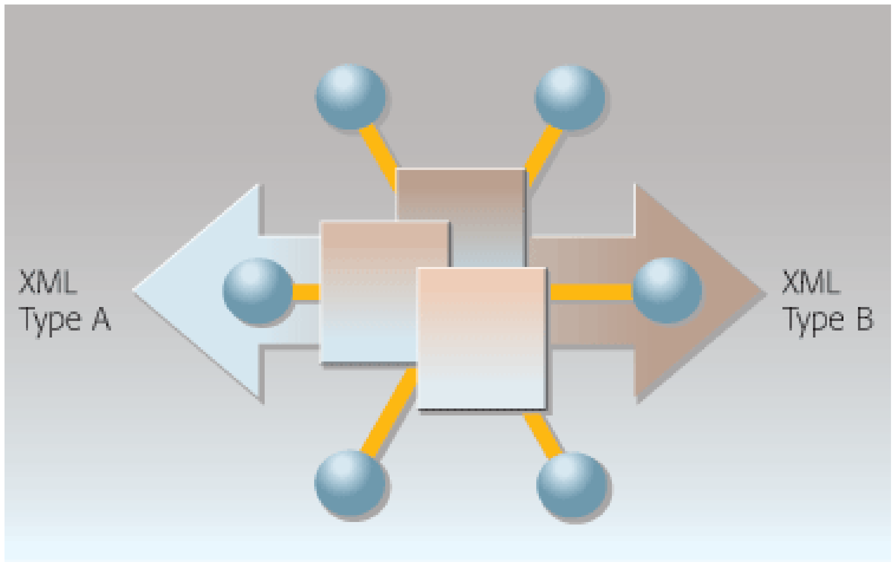


Figure 11
Service transformation pattern

11.5.6 Matchmaking Pattern

Another variant of the service routing pattern in which suitable target services are discovered dynamically based on a set of policy definitions (see Exhibit 11-10). This pattern is used in very dynamic environments where there are many hundreds or even thousands of services attached to the ESB, and service implementations may or may not be available when any given request is issued.

Exhibit 11-10: Matchmaking Pattern

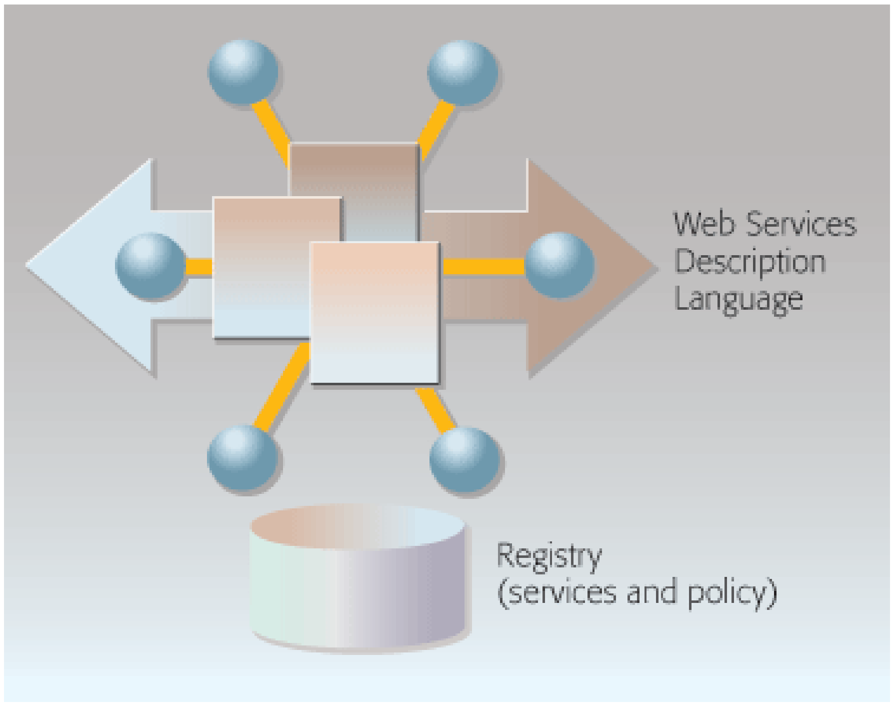

Web Services Description Language
Registry (services and policy)

Figure 12
Matchmaking pattern

These basic interaction patterns may also be used in conjunction with process-oriented interaction patterns. A process or workflow definition (defined by using BPEL or some equivalent language) extends the broker interaction pattern by orchestrating the execution sequence for a number of service interactions. By using these two patterns together, the service that orchestrates the interaction pattern can focus exclusively on business-process requirements, delegating issues of matchmaking, routing, and service selection to the ESB infrastructure.

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

[redacted]

- [redacted]
- [redacted]
- [redacted]
- [redacted]
- [redacted]

11.6.4 SOA Policy Management

When constructing an SOA, the notion of policy should be a starting point in understanding how an organization functions both internally and externally. [redacted] bottom-up.

Recommendations:

- SOA policy governance
- Compliance management and enforcement
- Safe and secure cryptography
- Application-level audits

[redacted]

11.6.5 Additional SOA Security Best Practices

[redacted]

[redacted]

11.7 Governance and Organization

An organizational and governance structure needs to be put in place to manage the SOA program. Governance should be structured around the following questions:

- Goals – What are the goals that SOA and particularly governance are trying to achieve?
- Stakeholders – Who will participate in the process? What are the roles and responsibilities?
- Processes – What are the processes, structures, organizational units, meeting, templates, etc. necessary for governance to operate efficiently?

The primary responsibilities of the governance are:

- Service Lifecycle Processes – Define processes for how to propose, implement, deploy, enhance, maintain, version and retire a service?
- Service Ownership – Define responsibilities for who owns (has responsibility for) a service?
- Service Funding – Define mechanisms for how service lifecycle phases funded?

- Decision and Issue Resolution – Define who is responsible for making certain decisions? How are issues resolved? How are issues escalated and appealed?
- Conformance – Define processes for how conformance to standards (design, technical) is verified?

The thrust of governance should be to help applications conform to the architecture. Experience has shown that a proactive approach is generally more effective than a purely compliance based approach. This can be accomplished through the use of standard templates and conformance questionnaires. But, DOI can go much farther than that. Providing examples of application design that conform to the architecture and standards is one approach. The most effective approach is to provide application architecture assistance to projects to help them during the design phase to understand the architecture and develop conforming designs.

Last but not least, the thorny issue of Funding must be resolved.

- Who pays for service development?
- Who pays for enhancements?
- Who pays for maintenance and operations?

Although there is not a simple answer to funding, experience has shown that complex, charge back methods are generally not that successful, especially during the initial rollout phases of an SOA. If at all possible, the cost for the services should be covered by a central group. Initially, use of services should be free to encourage adoption of SOA.

11.8 SOA Governance Best Practices[6]

Industry leaders have found that to improve the successful implementation of SOA, there is a requirement to develop SOA Governance early in the process. SOA services require improved governance to maintain the level of control needed to support the new Business/IT joint environment. The values provided by SOA Governance are:

- Realization of the business benefits of SOA
- Provides business process flexibility
- Improved time to market
- Mitigation of business risk
- Maintaining quality of service
- Consistency of service

SOA governance enforces the use of discipline to maintain consistency and relevance within the SOA life cycle. SOA governance tries to bridge the gap between business and IT by allowing traceability from business goals down to services and key performance indicators (KPIs) for measuring the results of those services. SOA governance also needs to keep a constant

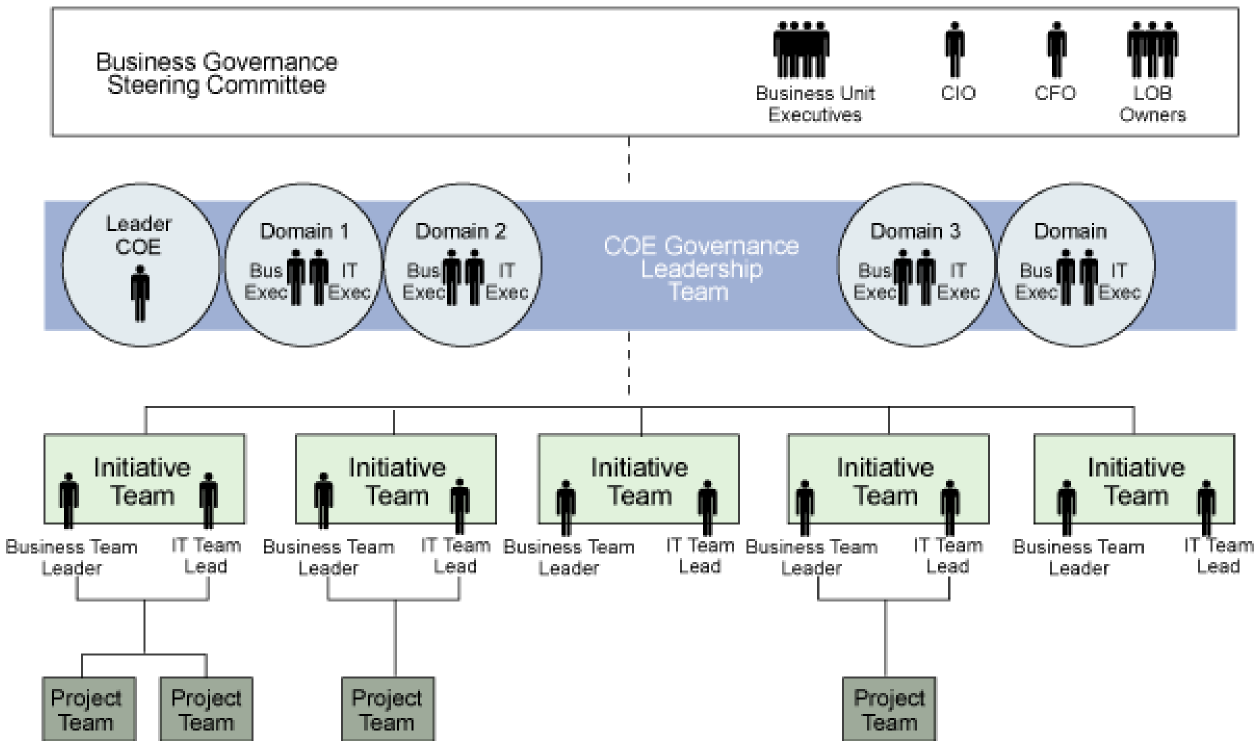
[6] http://www-306.ibm.com/software/solutions/soa/gov/

connection between business and IT through the concept of domain ownership. It is the responsibility of the members of the SOA governance council to logically partition the enterprise into a set of managed business services that share a common business context. Business owners and IT owners of a business domain are responsible for maintaining the applications that support the business domain's exposed business services. They are also responsible for maintaining and monitoring the SLAs of their existing business services as well as negotiating SLAs between different domains. The provisioning of metadata for enterprise business services is critical to both business and IT users. The metadata can provide information like WS-* compliance, business criticality, and so on. Based on the metadata, the business services can be monitored and managed. This is also a key responsibility of the members of the SOA governance council.

To ensure that services are not redundant and that they are relevant to business goals across the organization, the governance body should enforce coordination between new services and the existing services across the organization. This can be done by conducting periodic workshops with the LOB stakeholders to identify business application needs; after proper analysis, the governance body can add the business needs to the candidate business requirement portfolio. This can be followed by a series of business value assessment workshops wherein the identified candidates are passed through a business value indicator (BVI) litmus test to qualify a candidate business requirement as a service to be subsequently implemented and maintained.

The governance body is empowered with the responsibility of developing IT policies and oversees its compliance in the business applications that are designed and implemented. It should be a continuous exercise for the governance body to identify business processes that are critical either from a strategic differentiator perspective or for business process consolidation and optimization, or even just to stay competitive in the market. Any implementation of governance should be centered on the four pillars of an enterprise architecture: people, processes, technology, and services. One mechanism to implement an enterprise IT and SOA governance is by establishing a center of excellence (CoE) for IT and SOA governance that would enable a shared resource and capability center to function as a resource pool as new business application needs arise.

Exhibit 11-11: SOA Governance



A governance implementation needs to be supported by a hierarchical organizational reporting structure. As shown in Exhibit 11-11 above, such a reporting structure can be categorized into the four following hierarchies.

- **Sponsorship level**. This essentially consists of the stakeholders in the steering committee and is adequately represented by the members of the c-suite along with the LOB owners and executives. The steering committee articulates the business strategy, goal, and vision for the enterprise. Members of this level are the key decision makers on how IT investment needs to be made and channeled to specific areas of the business that either need business process improvement or need to implement new applications that can be competitive market differentiators.
- **Leadership level**. This is composed of the leader(s) of the governance CoE and two representatives (one business and one IT) from each business domain. (Note: Business domains as mentioned in the previous section represent a logical grouping of business services that share a common business context). The leadership team learns the business strategies and visions from the sponsorship members and also obtains directives from and reports to the steering committee. The leadership team creates enterprise IT architecture and SOA principles that stand as over-arching rules which any application architecture needs to conform to. The team also prioritizes which application architecture needs to be created and ensures that the IT priorities are aligned with the business needs. The governance body (represented by the leadership team) also documents the architecture standards and the compliance requirements to regulatory acts. The enterprise architecture constraints are also documented by this team, and the team is empowered with overseeing

the overall compliance to the architecture standards, guidelines, principles, and constraints when any new application needs to be designed and implemented (by teams at the next tier going down).

- **Opportunity management level**. Separate teams are formed at this level each focusing on one or more (related) business needs and are responsible to come up with clear definitions of business applications that cater to a given enterprise business need. Each initiative team has a business team lead responsible for gathering and formalizing the business requirements. Corresponding IT team leads are responsible for creating the overall application architecture and the solution that adheres to the IT and SOA principles mandated by the governance leadership team.
- **Project Management level**. Teams at this level manage the entire life cycle of a typical application design and development through the phases of solution definition, solution outline, macro design, micro design, build, test, and deploy. Each project team is aligned with a given initiative team. It is very common to have multiple simultaneous projects being run under a given initiative team.

11.9 DOI Timeline

TBD – recommended timeline for the steps listed in Section 11.1.